O trabalho apresentado nesta dissertação foi suportado pela Fundação para a Ciência e Tecnologia (FCT) através das Bolsas de Doutoramento com referência SFRH/BDE/33300/2008 e SFRH/BD/66242/2009

Filipe José de Oliveira Campos

Fault Tolerant Service Integration

UMinho | 2016

Universidade do Minho
Escola de Engenharia

Filipe José de Oliveira Campos

Fault Tolerant Service Integration

The MAP-i Doctoral Programme in Informatics, of the Universities of Minho, Aveiro and Porto

universidade de aveiro

Universidade do Minho

U.PORTO

June 2016

Universidade do Minho
Escola de Engenharia

Filipe José de Oliveira Campos

Fault Tolerant Service Integration

The MAP-i Doctoral Programme in Informatics, of the Universities of Minho, Aveiro and Porto

Universidade do Minho

supervisor:
Prof. José Orlando Pereira

June 2016

STATEMENT OF INTEGRITY

I hereby declare having conducted my thesis with integrity. I confirm that I have not used plagiarism or any form of falsification of results in the process of the thesis elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, 9th JUNE, 2016

Full name: FILIPE JOSÉ DE OLIVEIRA CAMPOS

Signature: Filipe José de Oliveira Campos

Agradecimentos

Foi uma caminhada mais longa que o esperado, mas contudo terminou. Nunca só nem mal acompanhado, foi da forma que me senti neste percurso, mesmo nos momentos mais difíceis, e por tal tenho que agradecer a todas as pessoas que de uma forma ou de outra me ajudaram.

Em primeiro lugar tenho que agradecer à minha querida esposa Ana por me ter sempre apoiado e ajudado a manter o nosso lar equilibrado, com os nossos filhos João e Tomás sempre a alegrar-me de forma a continuar até ao destino definido. Aos meus pais Ana e Francisco, e aos meus sogros Isabel e Claude, que sempre me auxiliaram, fazendo com que as dificuldades sentidas neste percurso fossem ultrapassadas mais facilmente.

Agradeço ao Professor José Orlando Pereira pela paciência, disponibilidade e acompanhamento dedicados neste percurso.

Agradeço ao Professor Rui Oliveira por tudo o que fez para nos proporcionar as melhores condições de trabalho possíveis.

I would like to thank Professor Karl Göschka for his valuable help and insight on the research work.

Quero agradecer a todos os que passaram pelo laboratório de Sistemas Distribuídos e pelo grupo OsSemEstatuto, e sem nenhuma ordem em particular: Ana Nunes, André Ferreira, Francisco Maia, Francisco Cruz, João Paulo, Miguel Matos, Ricardo Vilaça, Paulo Jesus, Jácome Cunha, Nuno Carvalho, Fábio Coelho, Nelson Gonçalves, Pedro Gomes, Ricardo Gonçalves, Nuno Castro.

Deixo um agradecimento especial aos meus amigos, Aníbal Pinto, César Freitas, Hélder Gomes, João Salgueirinho, Francisco Fernandes, João Vaz, Vítor Pinheiro, Luís Fernandes e Rómulo Gonçalves, por me apoiarem sempre, apesar da distância física de alguns deles.

Agradeço também a todo o HASLab (High Assurance Software Laboratory) e em particular ao GSD (Grupo de Sistemas Distribuídos) pelo ambiente fantástico criado entre todos. Um agradecimento especial ao David Rua da USE (Unidade de Sistemas de Energia) do INESC TEC (Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência) pelo auxílio

na ambientação às Smart Grids e respectivo funcionamento. Por fim, deixo também os meus agradecimentos a Francisco Lobo, Ilídio Martins, Adélio Fernandes e Sílvia Rocha, e a todos os que me apoiaram na Qimonda Portugal, S.A. no início deste percurso.

Ficam também os agradecimentos às instituições que apoiaram a execução do trabalho de investigação apresentada nesta dissertação. À Fundação para a Ciência e Tecnologia (FCT) que apoiou este trabalho através da bolsas de doutoramento (SFRH/BDE/33300/2008 e SFRH/BD/66242/2009). Este trabalho foi também apoiado pelo Projecto BEST CASE (NORTE-07-0124-FEDER-000056), financiado pelo Programa Operacional Regional do Norte (ON.2 – O Novo Norte) e pelo Quadro de Referência Estratégica Nacional (QREN) através do Fundo Europeu de Desenvolvimento Regional (FEDER), e por Fundos Nacionais através da Fundação para a Ciência e a Tecnologia (FCT). Ao INESC TEC e ao HASLab que apoiaram este trabalho através da bolsa de investigação inserida no projecto europeu 'CoherentPaaS: A Coherent and Rich PaaS with a Common Programming Model (FP7-611068)', oferecendo-me, juntamente com o Departamento de Informática da Universidade do Minho, as condições necessárias para o desenvolvimento deste trabalho.

Fault Tolerant Service Integration

Service Oriented Architectures (SOA) are a mainstay of enterprise computing and there is now a growing interest in services for systems of connected devices in a variety of environments, ranging from industrial manufacturing equipment to home automation, and other highly heterogeneous environments. In fact, the current trend in connected devices is expected to accelerate as the vision for the Internet-of-Things (IoT) becomes a reality. The IoT embodies the seamless discovery, configuration, and interoperability of networked devices in various settings, and in a sense, it has extended the application range of Enterprise Application Integration (EAI) to non enterprise environments. For instance, EAI in manufacturing environments with highly demanding dependability and timeliness requirements, must leverage closed proprietary middleware solutions that incorporate some fault tolerance techniques to fulfill such requirements, since transactional processing does not satisfy those requirements completely.

But as non enterprise applications become increasingly critical, the middleware coping with Machine-to-Machine (M2M) communication and coordination, such as the Devices Profile for Web Services (DPWS), has to deal with fault tolerance and increasing complexity, while still abiding to resource constraints of target devices. Fault tolerant service integration in such scenarios can then be broken down into reliable communication and service correctness. These two features can be provided by gossip protocols, to ensure reliable message exchanges in different communication patterns, and consensus protocols, to ensure the normal behavior of intervening services.

In this dissertation, we address these challenges by proposing a DPWS-based framework containing a gossip service and a consensus service, and evaluate its effectiveness on providing fault tolerance capabilities to existing services.

Integração de Serviços com Tolerância a Faltas

As Arquiteturas Orientadas a Serviços (*SOA*) são um dos pilares da computação empresarial e há atualmente um interesse crescente na utilização de serviços para sistemas com dispositivos ligados numa variedade de ambientes, que vão desde a produção industrial à domótica, até outros ambientes altamente heterogéneos. De facto, a atual tendência em dispositivos ligados deverá acelerar à medida que a visão da Internet das Coisas se torne uma realidade. A Internet das Coisas incorpora descoberta automática, configuração e interoperabilidade dos dispositivos ligados em rede em vários ambientes, e em certo sentido, ampliou o alcance de aplicação de *Enterprise Application Integration* (*EAI*) até ambientes não empresariais. Por exemplo, *EAI* em ambientes de produção com exigências de fiabilidade e pontualidade altamente exigentes, deve alavancar soluções de middleware proprietário que incorporem algumas técnicas de tolerância a faltas para cumprir esses requisitos, uma vez que o processamento com recurso a transações não os preenche completamente.

À medida que aplicações não empresariais se tornam cada vez mais críticas, o middleware que lida com a comunicação e coordenação Máquina-a-Máquina, como *Devices Profile for Web Services* (*DPWS*), tem de lidar com a tolerância a faltas e o aumento da complexidade, respeitando simultaneamente as limitações de recursos dos dispositivos alvo. A integração de serviços com tolerância a faltas em tais situações pode, então, ser dividida em comunicação confiável e correção dos serviços. Estas duas características podem ser fornecidas por protocolos epidémicos, de forma a garantir a troca fiável de mensagens utilizando diferentes padrões de comunicação, e por protocolos de consenso, para garantir o normal funcionamento dos serviços intervenientes.

Nesta dissertação, estes desafios foram abordados com uma infraestrutura baseada em *DPWS* que inclui um serviço epidémico e um serviço de consenso, tendo sido avaliada a sua eficácia a assegurar tolerância a faltas nos serviços existentes.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The growing interest in Service-Oriented Computing (SOC) and Enterprise Application Integration (EAI) in a diversity of environments has translated into a number of widely accepted standards ranging from data formats and the messaging infrastructure to a standard services portfolio.

A particularly interesting area of research and development is the coordination and composition of multiple services, or multiple instances of services, to achieve complex behaviors or provide additional guarantees. For instance, service coordination using the ubiquitous transaction processing paradigm has been addressed by WS-AtomicTransaction and WS-BusinessActivity, which build upon the WS-Coordination specification in order to coordinate Web Services. WS-AtomicTransaction provides the type of coordination of atomic transactions used in applications that require consistent agreement on the outcome of short-lived distributed activities that have the all-or-nothing property. WS-BusinessActivity provides the coordination type for business activities that require consistent agreement on the outcome of long-running distributed activities. WS-Coordination specifies an extensible coordination framework for Web Services, which features some interesting properties that allow them to achieve reliable interactions. The remaining standards for Web Services that are specially focused on fault tolerance include WS-ReliableMessaging and WS-Reliability, which provide reliable communication only between two points, hence not enabling a multicast-typed communication.

On the other hand, there are several protocols that provide other fault tolerance mechanisms for Web Services, but since they are not standardized, the inclusion of fault tolerant techniques in building a dependable and in-

teroperable architecture for service integration might be limited to reliable point-to-point communication and transactions, due to the specific nature of different services.

However, service integration is needed in a variety of scenarios with widely diverse requirements. For instance, production floor and healthcare environments, as well as smart grids, have stringent dependability and timeliness requirements that are not entirely satisfied by, or even compatible with transactional processing. On the other hand, a smart house environment, where a wide range of increasingly intelligent devices coexist, has normally lax requirements, which transactional processing can possibly match, but the inherent consumption of resources can overwhelm the capabilities of such devices.

A few attempts to address this gap exist [He, 2004; Osrael et al., 2007b; Salas et al., 2006], but only cover a very small subset of fault tolerance techniques and fall short as a general interoperability solution. The main challenges arise from the complexity of many fault-tolerant solutions, such as a view synchronous group communication protocol, but also from the subtle impact of service decomposition on the assumptions of such algorithms.

In fact, one can even consider decomposing the major building blocks of group communication themselves, namely, gossip-based dissemination and consensus protocols. Gossip and consensus protocols have interesting properties, like scalability and agreement, which can prove to be useful to improve the fault tolerance capabilities of Web Services, more concretely in service coordination scenarios.

A gossip-based communication protocol is inspired by the form of gossip in social networks, and also in the way viruses spread in a biological community, hence also being known as epidemic protocols. This kind of protocols provides a way to spread messages to a whole system and also to process acknowledgments in a distributed fashion, in order to avoid network congestion and nodes to be flooded with acknowledgments, or even to totally avoid this processing due to its inherent all or nothing message delivery guarantees which enable its usage for reliable multicast, specially adequate in settings comprising a large amount of nodes that must communicate. At a lower level, gossip can be used by a consensus protocol to provide communication among all the nodes involved in the decision. And at an higher level, it can be used to implement, for instance, a membership service [Vogels and Re, 2003].

The consensus problem can be defined as the agreement on a set of processes to decide on a common value even if each of them starts with a different value. It corresponds to an abstraction of the problem of all processes in a fault-tolerant distributed system agreeing on the same value despite having started with different opinions [Pereira and Oliveira, 2004]. A generic con-

sensus service [Guerraoui and Schiper, 2001] would be extremely useful for building fault-tolerant agreement protocols which could be used in a Web Services environment according to its resources and capabilities, and the required usage for a consensus mechanism.

Being consensus the basic problem involved in fault tolerance, since it involves defining whether a process or node is up or down, and since gossip protocols are used to achieve reliable multicast, they can be combined to build a large variety of distributed systems.

1.2 Problem statement

Service integration is needed in a variety of scenarios with widely diverse requirements, from smart grids, with very strict requirements, to smart houses, where a wide range of heterogeneous devices coexist. The Devices Profile for Web Services (DPWS) defines a set of protocols that resource constrained devices should implement in order to achieve seamless networking and interoperability through Web Services, enabling the interaction between devices from such disparate environments. Since DPWS does not provide any fault tolerance mechanism, operations are susceptible to both communication and service faults and cannot be restarted or recovered if stopped. This is particularly worrisome as notifications and configuration updates may correspond to critical alerts and urgent commands.

Thus, scalable lightweight coordination and replication protocols that ensure service dependability, while fitting the general DPWS assumptions, are necessary. At the same time, the application of these protocols must not affect the modularity and interoperability of the existing services, while proving their usefulness in needing scenarios.

1.3 Approach

In an increasingly Service Oriented Technological World, standards must be defined in order to achieve interoperability among the various heterogeneous systems in existence. To that end, new advances should try to take advantage of the existing standards, and aim at producing effectively useful protocols that attract the interest of companies, and, consequently, of standardizing consortiums or entities.

Our approach to solve this problem is to extend existing Service Oriented Architectures with services that provide fault tolerance and reliability guarantees to the existing services in a transparent way. In that sense, we first

aim at gossip and consensus protocols composed with basic services that, in some way, correspond to the building blocks of those types of protocols and do not provide any fault tolerance guarantee by themselves. The structure of these basic services, should provide the required functionality of the represented feature that allows the several variations of these protocols to be produced, through some changes in configuration or parameterization.

Building more complex services, that somehow rely on gossip or consensus protocols to obtain some fault tolerance guarantee, with the resulting services will then be possible. It is expected that this higher level services will be able to provide the desired fault tolerance guarantees.

As the standard technology used to implement Service Oriented Architectures, Web Services contribute greatly to better perceive the composition and integration of services. However, this technology may not be suitable for a production environment, mostly due to the timeliness requirements of the systems in existence in this kind of scenarios. To produce an adequate solution, a new system can be developed, integrating the lessons learned during this research project.

1.4 Case study

The Smart Grid vision embodies the future power grid, which aims to increasingly integrate renewables as well as promoting the generalized participation of different entities, to achieve better operation reliability, by tolerating power failures, and efficiency, through the reduction of carbon emissions and fuel costs, transmission losses, and deferral of investments, among others. To fulfill this vision, some important challenges in interoperability, reliability, and scalability need to be addressed. As the sheer scale of the electric grid and the criticality of the communication among its subsystems for proper management, demands a scalable and reliable communication framework able to work in an heterogeneous and dynamic environment. Moreover, the need to provide full interoperability between diverse current and future energy and non-energy systems, along with seamless discovery and configuration of a large variety of networked devices, ranging from the resource constrained sensing devices to servers in data centers, requires an implementation-agnostic Service Oriented Architecture.

To overcome these challenges, the usage of the proposed framework, that reconciles the reliability and scalability of Peer-to-Peer systems, with the industrial standard interoperability of Web Services, adding also data aggregation capabilities, was illustrated in three specific scenarios. The first scenario shows how the framework can replace the existing mechanisms of alert and

event propagation in the Automated Metering Infrastructure (AMI) for environments with a high rate of messages and a large number of targets. The second scenario demonstrates the ability of the proposed framework to collect metrics from different points of the Smart Grid in order to plan power production according to the announced energy requirements. The third scenario illustrates how critical configurations can be performed in various instances of a service, while ensuring its availability in multiple points of the Smart Grid.

1.5 Objectives and results

The goal of this project is to advance the state of art in service integration in large-scale distributed and heterogeneous systems with strict dependability requirements by:

- Leveraging existing fault-tolerant protocols in a Service Oriented Computing context. This requires:
  - Matching theoretical assumptions to actual environments;
  - Assessing the feasibility and adequacy of known protocols;
  - Decomposing them into interoperable services.
- Addressing the impairments to fault tolerance arising from Service Oriented Computing, namely, from composition strategies and very large number of components. This requires precise characterization of problems, and proposal of novel protocols to address them.

Specifically, the expected result of this project is a framework of Web Services that provides transparently fault tolerance and dependability guarantees to existing services.

1.6 Contributions

The main contribution of this research is to increase the reliability of lightweight middleware architectures, namely Web Services which have become available on resource constrained devices, through the Devices Profile for Web Services (DPWS). This contribution can be broken down into two specific features:

- WS-Gossip enables reliable communication using gossip protocols on Web Services, while providing useful message exchange patterns such

as many-to-one or one-to-may on heterogeneous environments, where Web Services reliable messaging standards are limited to point-to-point communication and show interoperability issues.

- Raft4WS enables the usage of the Raft consensus protocol on Web Services, more precisely on DPWS, allowing the usage of fail-crash resilient replicated services in such lightweight scenarios.

1.7 Publications

- An experimental evaluation of machine-to-machine coordination middleware.
  Filipe Campos and José Pereira.
  In the 30th Annual ACM Symposium on Applied Computing (SAC). ACM, 2015.

- An experimental evaluation of machine-to-machine coordination middleware: Extended version.
  Filipe Campos and José Pereira.
  arXiv, cs.DC, Dec 2014. 24 pages, Technical Report.

- A peer-to-peer service architecture for the Smart Grid.
  Filipe Campos, Miguel Matos, José Pereira and David Rua.
  In IEEE Fourteenth International Conference on Peer-to-Peer Computing (P2P). IEEE, 2014.

- Coordenação de Serviços Web heterogéneos com tolerância a faltas.
  Filipe Campos, Miguel Matos and José Pereira.
  In INForum 2014 - Atas do 6º Simpósio de Informática. FEUP Edições, 2014.

- Improving the Scalability of DPWS-Based Networked Infrastructures.
  Filipe Campos and José Pereira.
  arXiv, cs.DC, July 2014. 28 pages, Technical Report.

- Experimental Evaluation of Distributed Middleware with a Virtualized Java Environment.
  Nuno Carvalho, João Bordalo, Filipe Campos and José Pereira.
  In MW4SOC' 11: Proceedings of the 6th workshop on Middleware for Service Oriented Computing. ACM, 2011.

- Achieving eventual leader election in WS-Discovery.
  Filipe Campos, José Pereira and Rui Oliveira.
  In Proceedings of 5th Latin-American Symposium on Dependable Computing (LADC). INPE, 2011.

- Gossip-based service coordination for scalability and resilience.
  Filipe Campos and José Pereira.
  In MW4SOC '08: Proceedings of the 3rd Workshop on Middleware for Service Oriented Computing. ACM, 2008.

- WS-Gossip: Middleware for scalable service coordination.
  Filipe Campos and José Pereira.
  In Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion. ACM, 2008.

1.8 Availability of code

The source code developed and used for all the experiments in this thesis is available as open source, allowing the experiments to be reproduced.

Our implementation of WS-Gossip on the WS4D stack is available at `https://github.com/filipecampos/ws_gossip`, and the code used in the WS-E, WS-RM and Multicast scenarios, as well as for setting up and controlling all the experiments, is available at `https://github.com/filipecampos/ws_gossip_tests`.

Our implementation of Raft on the WS4D stack, that can be used to build applications, is available at `https://github.com/filipecampos/raft4ws`. The code used for setting up and controlling the experiments is available at `https://github.com/filipecampos/raft_tests`.

Chapter 2

State-of-the-Art

2.1 Devices Profile for Web Services

The Devices Profile for Web Services (DPWS) [DPWS] defines a set of protocols that resource constrained devices should implement in order to achieve seamless networking and interoperability through Web Services. It assumes that each device behaves as a standard *hosting service*, providing basal functionality, and exposing one or more *hosted services* that offer device specific functionality. Besides basic SOAP, WSDL, the HTTP binding, WS-Addressing, and WS-Security, that are at the core of Web Services capabilities and interoperability, the DPWS protocol stack covers the following areas:

- SOAP-over-UDP [SOAP-over-UDP] binding provides a lightweight protocol for network interactions that don't need the flexibility of the full HTTP stack, namely, regarding the amount of data that can be transferred, and support network level multicast, thus paving the way for dynamic discovery.

- Dynamic discovery is supported by combining WS-Discovery [WS-DD], WS-MetadataExchange [WS-ME] and WS-Policy [WS-P]. Together, they allow a client: to discover devices in the network, according to their identification or exported services; to list resources contained within the scope of a discovered device; and to learn about their characteristics and non-functional requirements.

- Simple publish/subscribe communication through WS-Eventing [WS-E], for instance, allowing services to notify interested clients on changes to monitored resources.

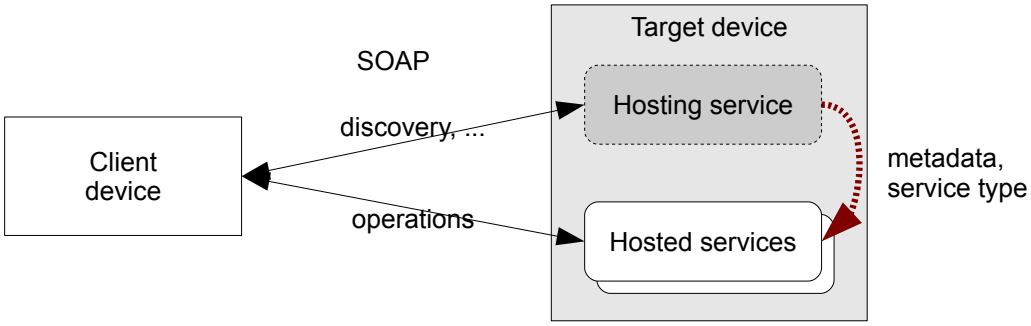


Figure 2.1: Overview of DPWS architecture.

Consider the following example of a typical DPWS deployment scenario, depicted in Figure 2.1. White boxes denote custom components, while gray boxes denote infrastructure components provided by the selected DPWS toolkit. An interaction starts when a client probes for a device or service, or when a device, upon connection, advertises its presence. This is achieved through WS-Discovery by using multicast SOAP-over-UDP messages. If a device that supports the Discovery Proxy role, defined in WS-Discovery, is present in the network, it keeps a registry of all available devices and allows most multicast messages to be suppressed.

Upon discovery, the client obtains a description of the resources available in the device, kept as an internal reference to *hosted services* (shown as a red dotted arrow). The client can then invoke *hosted services* directly, with no wrapping or indirection, or manipulate resources directly by requesting and transferring their XML representations. Moreover, the client can subscribe to notifications from services that expose such functionality, thus being able to avoid polling for changes, since it receives asynchronous notifications.

There are multiple implementations of the DPWS in various programming languages. Namely, the Web Services for Devices (WS4D) project [WS4D] provides both C and Java implementations. In particular, the WS4D Java Multi Edition DPWS Stack (JMEDS) supports a wide range of devices as it is compatible with both the Standard and Micro Editions of the Java platform. Modern operating systems, such as Windows Vista, Windows Embedded CE, and Windows 7 are shipped with a built-in DPWS framework, thus rendering this specification available in most personal computers and in many devices such as set-top boxes.

2.1.1 Limitations of DPWS

Although DPWS provides an adequate infrastructure for small scale systems, such as home automation, it is becoming increasingly interesting when managing large number of components. DPWS has however some scale limitations. First, using notifications based on WS-Eventing [WS-E] imposes a burden on the device that is being observed, that has to issue notifications to all observers. Moreover, when a resource exposed by many devices has to be updated, *e.g.* to change a configuration variable, the initiator device has to contact all destinations one by one. Finally, as there is no support for transactional coordination mechanisms, such lengthy operations involving large numbers of destinations are susceptible to faults and cannot be restarted or recovered if stopped. This is particularly worrisome as such notifications and configuration updates may correspond to critical alerts and urgent commands.

Note that it does not make sense to resort to heavyweight coordination protocols such as WS-Coordination [WS-C] and WS-AtomicTransaction [WS-AT] or even to a hierarchical structure comprised of 'superpeers', which would introduce a dependability on some special, and most certainly more powerful, machines that might not be available in most scenarios. Even if devices could support their requirements, for instance, in terms of stable storage, these protocols would also not scale to hundreds of participants. Thus, a scalable lightweight coordination protocol, that fits the general DPWS assumptions, is necessary.

2.1.2 WS-Eventing in detail

The WS-Eventing specification supports simple publisher-subscriber interaction, by defining how Web Services can subscribe to or accept subscriptions for event notification messages [WS-E].

This specification defines the four following roles:

Event Source Sends notifications on triggered events, and accepts requests for creating subscriptions.

Subscription Manager Manages event subscriptions. It notifies *Subscribers* when their subscriptions are terminated unexpectedly, and replies to their subscription management enquiries, such as subscription's status retrieval, renewal or deletion.

Event Sink Receives event notification messages.

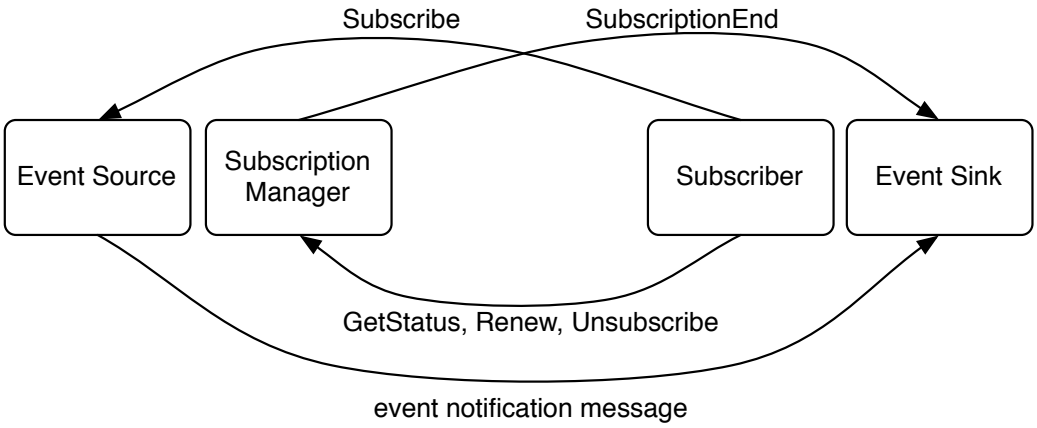


Figure 2.2: WS-Eventing components.

Subscriber Contacts an *Event Source* to create a subscription to manifest the interest of its associated *Event Sink* to be notified on the occurrence of some event. It is also responsible for issuing subscription management requests to the *Subscription Manager*.

In the simplest scenario, with only two intervening Web Services, a publisher will comprise both the *Event Source* and the *Subscription Manager* roles, whereas a subscriber will accumulate both the *Subscriber* and *Event Sink* roles.

WS-Eventing defines the concept of Delivery Mode, in order to better adapt the general publish-subscribe pattern to scenarios with different event delivery requirements. The default delivery mode of this specification is the single delivery asynchronous Push mode. But, for instance, situations with slow event consumers where they poll for event messages may be preferred, in order to control the rate of message arrival and to avoid overwhelming them if the rate of generation and transmission of event messages is far superior to their processing rate. The Subscribe request message defines the specific delivery mode to be used in notifying the identified *Event Sink*, and new delivery modes can be freely used, if both publishers and subscribers support them. In the event that a *Subscriber* requests a delivery mode that is not supported by the *Event Source*, it will respond signaling this situation, and it may convey a list of the supported delivery modes. This specification also allows notifications to be wrapped in a standard message, instead of the default unwrapped mode where each notification is transmitted as a message typed according to the event's action.

Although it lacks explicit support for brokered dissemination, it embodies a flexible filtering mechanism in the base specification, favoring lightweight

implementations and many-to-one dissemination scenarios. And since it was backed by major vendors, such as IBM, Microsoft or TIBCO, it has therefore been the preferred choice for connected devices, namely, within WS-Management and DPWS. The alternative family of standards embodied in OASIS WS-Notification, which besides providing simple notification and subscription mechanisms, also provides extensible topic definition and brokered dissemination.

2.2 Service coordination

Ever since the first use of transaction processing, there have been a variety of transaction protocol standards, such as X/Open and the OTS, and vendor-specific protocols, with many corresponding implementations. Interoperability between these incompatible protocols has always proved problematic and there has been limited success, but Web Services offer a solution to this problem [Little and Freund, 2003].

However, there is a paradox associated with Web Services, since they provide a service oriented, loosely coupled, and potentially asynchronous means of propagating information between parties, while the underlying services use traditional transaction processing infrastructures.

A transaction provides a mechanism for grouping multiple Web Services operations into a larger unit of work whose results can be coordinated to achieve overall success or failure. A variety of transaction protocols are available for use with different types of applications, ranging from tightly to loosely coupled, and from short-term interactions to long-running business process executions. Choosing the right protocol for the application helps ensure that composite Web Services applications can achieve consistent, predictable, and reliable results.

2.2.1 Definition

Service coordination has been defined as 'how services work together to fulfill a business process' [Margolis and Sharpe, 2007]. A business process models a business flow that may span through several phases, departments, and may include several business tasks. Migrating the actual business processes of an organization to a Service Oriented Architecture (SOA) infrastructure, implies that several applications inside the organization are automated, for the provided services to be used by the processes [Juric et al., 2007]. A business process defines which services must interact, specifying the necessary conditions and inputs for the operations and the timings when they should

be invoked, to embody a certain business function. Hence, the organization of such workflows can be achieved by using coordination protocols.

Coordination protocols are used to support a number of applications, usually those that need to reach consistent agreement on the outcome of distributed activities, such as reaching consensus on a decision like a distributed transaction, or providing the guarantee that all participants obtain a specific message in a reliable multicast environment. The distributed activities normally require central coordination and housekeeping, and could have different purposes such as transaction coordination, audit logging, etc.

Statelessness is a key service-orientation principle, which coordination protocols reinforce by assuming responsibility for the management of context information, alleviating the need for services to retain state.

Service coordination encloses an important feature, transaction control which can be viewed as the way services handle the changes resulting from their activities. These changes can be *committed*, ensuring that they become permanent, or *rolled back*, erasing the current modifications, or even *compensated* by some service responsible for reversing the effects of the changes in some way.

2.2.2 Types of service coordination

Service coordination can take two different forms: orchestration or choreography [Margolis and Sharpe, 2007]. Orchestration is a form of processing where there is a service that behaves just like a *maestro* in an *orchestra*, coordinating the actions of all the services, which can be viewed as instrumental groups using the same musical *orchestra* analogy, according to the received messages. It is often viewed as the coordination of services within a company. Choreography is a more decentralized form of processing where there is no central point of coordination and the interaction of the services is based on rules known to the services, that can be specific to each one of them. This mechanism is often viewed as the coordination of the activities of trading partners. These two types of service coordination are often used interchangeably by some analysts which leads to some confusion.

The well-known *2 Phase Commit (2PC)* mechanism does not suffice for the Web Services context due to its cross-enterprise nature and to the lack of trust between parties when interactions occur across enterprise boundaries [Alonso et al., 2004]. Using a *2PC* Protocol that unlocks resources only at the end of its entire duration is not an appropriate solution for long-running business transactions among different trust domains.

To maintain a persistent activity in a Web Services environment differs from the traditional distributed environments due to its loosely coupled na-

ture. In this case, to preserve the integrity of an activity, a context management service is needed. In addition to this service, structured protocols are required to define behavioral characteristics of the services involved in the activity.

To standardize the context management service and protocols, some specifications for coordination of Web Services were developed. These specifications can be considered to define some form of orchestration, where there can be one or more central entities that control the interaction of the services.

One concept that will be used throughout this document, whose meaning should be well defined is the concept of activity. An activity is defined as a computational unit composed of a number of tasks, for which transaction coordination is required, carried out across one or more Web Services.

2.2.3 Types of protocols

There are two different kinds of coordination protocols: vertical and horizontal. Vertical protocols define more than just coordination-related protocols, for instance communication, and are specific to an industry or business area, reason why they are also called *business protocols*. They usually specify in more detail, than horizontal protocols, the way transactions should be performed, as well as the documents, their formats, semantics, contents and operations [Alonso et al., 2004]. Many of them have been extended to support XML and Web Services, as they precede the appearance of these technologies.

Horizontal protocols are generally useful and applicable in many business scenarios, like Web Services that are not specific to any particular sector or industry, and usually provide middleware features and properties, and therefore they may also be called *middleware protocols*.

The Electronic Business using eXtensible Markup Language (ebXML) [Wiki ebXML] is an horizontal standard specially aimed at *Business-to-Business (B2B)* interactions, and it provides a set of specifications common to the entire e-business sector.

Contrarily to ebXML, RosettaNet is a vertical standard, since it focuses on the business area of supply chain automation and optimization of a specific industry, electronic components manufacturers. There have been efforts for the convergence of these two standards, as well as others, which has resulted in the duplication of features [Badakhchani, 2004].

Another *B2B* standard that is worth mentioning is XML Common Business Library (xCBL) [Wiki xCBL], which defines the roles and documents involved in the order management protocols, as well as their use in the transactions. The approach of xCBL is to focus and standardize the most impor-

tant and used exchanges, such as order management, package and transport, and invoicing, among others [Alonso et al., 2004].

As these standards are not completely compatible with regular Web Services, more emphasis will be put on other standards that build upon Web Services.

OASIS Business Transactions Protocol (BTP)

The Business Transaction Protocol (BTP) was the first standard for XML transactions in loosely coupled domains such as Web Services. It was first developed by a consortium of companies such as Hewlett-Packard, Oracle and BEA, and then handed over to the OASIS Business Transactions Technical Committee, which approved and published the first and only existing version, 1.0 [BTP], in June 2002. From that moment on, a draft for version 1.1 was proposed in November 2004, but never ratified, which seems to have sealed the fate of this standard, as well as the closure of the committee on February 7, 2006 due to inactivity.

BTP supports applications which are disparate in time, location, and administration and, thus, require transactional support beyond classical ACID transactions. This protocol has two fundamental units:

Atom Uses a two-phase protocol, but without any guarantee of atomicity using strict two-phase locking. Ensures the same outcome for all participants.

Cohesion Has more relaxed atomicity and participants may get different outcomes according to business logic. Allows the transaction as a whole to make forward progress even in the event of failures.

The fact that both the WS-Coordination and WS-CAF specifications surpassed BTP, is explained recurring to several technical reasons, such as [McGovern et al., 2006]:

- the complexity of the protocol;
- the freedom given to implementors of the standard, leading to largely disparate behaviors;
- the fact that it was not designed solely for Web Services;
- the lack of immediate interoperability with existing transaction processing infrastructures.

Albeit all this reasons, it is very likely that it was mainly the lack of support from major vendors that lead to its fall.

WS-Coordination

The WS-Transaction family of specifications was developed by BEA, IBM, and Microsoft and released originally in conjunction with WS-BPEL in August 2002. These specifications were updated in September 2003 when WS-BusinessActivity was broken out into its own specification. The original WS-Transaction specification was also renamed WS-AtomicTransaction at that time.

The OASIS WS-Transaction Technical Committee is now responsible for defining both of these transaction protocols, as well as the basilar WS-Coordination standard. The latest version of all of these OASIS standards, 1.2, was published on February 2, 2009.

WS-Coordination [WS-C] specifies an extensible framework for *context* management, which provides coordination for the actions of distributed applications. This coordination is achieved through provided protocols that support distributed applications, for instance, those that need to reach consistent agreement on the outcome of distributed transactions.

An application service can create a *context* needed to propagate coordination information to other services involved in an activity. These services will then need to register as participants for the activity. For this purpose, the application must include the created *coordination context* in the messages that it sends to the referred services.

A *coordination context* can be transmitted using application-specific mechanisms, such as a header element of a SOAP application message. This kind of conveyance is commonly referred to as flowing the *context*.

The structure of a *context* and the requirements to propagate it between cooperating services are also defined in WS-Coordination, and can depend on the type of coordination that is used. A *coordination context* contains information on:

- how to access a coordination registration service;
- the coordination type;
- relevant extensions.

This framework also enables existing transaction processing, workflow, and other systems for coordination to hide their proprietary protocols and to operate in an heterogeneous environment.

This specification is not enough to coordinate Web Services, since it provides only a coordination framework, leaving the concrete protocol and targeted coordination type undefined. Two separate standards, WS-AtomicTransaction and WS-BusinessActivity, implement the WS-Coordination

framework, by defining their own coordination type: short-term atomic transactions, and long-running business activities, respectively.

When using WS-Coordination, a service, that is usually the starter of the interaction, interacts with a *coordinator*, that corresponds to the *maestro* in an orchestra. The *coordinator* hosts the following services:

Activation service creates and returns a *coordination context* when invoked. A *context* contains information, like the identification of the transaction and *coordinator*, that must be available to any service that participates in a coordination protocol. The exact semantics are defined in the specification that defines the coordination type. The *context* also defines the runtime existence of the activity and establishes a level of control on how the task in execution can be processed. After the creation of a *context*, the requester receives it from the activation service, and sends it enclosed in the invocations to other services so they can register for the coordinated activity.

Registration service allows a service that holds a *context* to register for a particular activity, hence becoming a participant in a coordination protocol. This participant becomes aware of the *Registration Endpoint Reference*, when it receives the *coordination context* enclosed in an application message. This service also allows an interposed *coordinator* to register for an activity in the name of an application. One effect of the registration is to provide the access details for one or more coordination-protocol services. After receiving a *register* message from a participant, that includes its protocol service *Endpoint Reference* as a parameter, the *coordinator* sends a response including its corresponding protocol service *Endpoint Reference*. After this step, both sides can exchange protocol messages, because they can target each other. A *Registration service* is not required to detect duplicate *register* requests and may reckon each of them as an attempt to register a distinct participant. If a participant sends multiple *register* requests for the same activity, it must handle correctly duplicate protocol messages from the *coordinator*. The manner in which the participant handles duplicate protocol messages depends on the specific coordination type and coordination protocol.

The coordination protocols and types are defined in other standards such as WS-AtomicTransaction, WS-BusinessActivity, or even WS-BPEL.

WS-AtomicTransaction (WS-AT) The WS-AT specification [WS-AT] defines a protocol that can be plugged into WS-Coordination to provide an

adaptation for Web Services of the classic *2PC* [Newcomer and Lomow, 2004] mechanism. However, it is often said that this mechanism does not adapt well to Web Services [Newcomer and Lomow, 2004]. Nonetheless, it is adequate for interoperability across short-lived, co-located services that need to ensure consistent, all-or-nothing results for a transaction.

The purpose of WS-AT is to handle a *2PC* process to make the changes, resulting from the activity of some service, persistent. A *2PC* process consists on a poll conducted by the *coordinator* that will lead it to send two alternative directives to all the resource managers involved in the transaction:

- *commit*, if all of the registered services have responded indicating that the changes were successful;
- *rollback*, if at least one of the registered service fails to respond or responds indicating a failure.

A service that participates in an atomic transaction can register for more than one of the different types of coordination protocols, as defined in the WS-AT specification:

Completion initiates commit processing when an application tells the *coordinator* to either try to commit or abort an atomic transaction. Based on the registered participants for each protocol, the *coordinator* begins with Volatile 2PC and then proceeds through Durable 2PC. After the transaction has completed, a status is returned to the application and the final result to the service that initiates the transaction (*initiator*), if it has registered for this protocol.

Two-Phase Commit coordinates registered participants to reach a commit or abort decision, and ensures that all participants are informed of the final result. It has two variants:

Volatile 2PC where participants manage volatile resources such as a cache register or a window manager. Upon receiving a *Commit* notification in the Completion protocol, the *root coordinator* begins the *Prepare* phase of all participants registered for the Volatile 2PC protocol. All participants registered for this protocol must respond before a *Prepare* is issued to a participant registered for Durable 2PC. Further participants may register with the *coordinator* until it issues a *Prepare* to any durable participant. A volatile recipient is not guaranteed to receive a notification of the outcome of the transaction [Newcomer and Lomow, 2004].

Durable 2PC where participants manage durable resources such as a database register or a file. Upon successfully completing the *Prepare* phase for Volatile 2PC participants, the *root coordinator* begins the *Prepare* phase for Durable 2PC participants. All participants registered for this protocol must respond *Prepared* or *ReadOnly* before a *Commit* notification is issued to a participant registered for either protocol.

WS-BusinessActivity (WS-BA) The purpose of WS-BA [WS-BA] is to handle a relatively complex business interaction having, very often, a long time span, opposed to the direct nature and short duration of atomic transactions. The underlying transactional model for this specification is the so-called open nested transaction as mentioned in [Cabrera and Kurt, 2005].

WS-BA allows *context* information to be preserved for long-running transactions, albeit resources are not locked, emphasizing the difference to the way atomic transactions are performed and also how *coordinators* of both types deal with activity failures. The business activity *coordinator* supplies a *compensation* process that is executed when failures occur in the original activity. A *compensation* is different from an exception handling routine, as it can generate a process activity of its own, which, by itself, may require separate exception handling.

Contrarily to atomic transactions, where resources involved in an activity remain locked, typically for a short period of time, until the transaction finishes, a business activity can take a long period of time to run which could starve a system if its resources were locked.

The WS-BA specification provides the definition of two business activity coordination types:

AtomicOutcome *Coordinators* must direct all participants either to close or to compensate.

MixedOutcome *Coordinators* must direct all participants to an outcome but may direct each individual participant to close or compensate.

All business activity *coordinators* must implement the AtomicOutcome coordination type but it is not mandatory for them to implement the MixedOutcome coordination type.

This specification also defines two specific business activity agreement coordination protocols:

BusinessAgreementWithParticipantCompletion A participant registers for this protocol with its *coordinator*, and knows when it has completed all work for a business activity.

BusinessAgreementWithCoordinatorCompletion A participant registers for this protocol with its *coordinator*, and relies on it to be informed when all the requests to perform work within the business activity have been received.

These protocols ensure consistent agreement on the outcome of long-running distributed activities, and they may be combined with either business activity coordination types.

In the case of the BusinessAgreementWithParticipantCompletion protocol, all the participants know when each of them has completed all work for a business activity and so they all trigger the coordination process, contrarily to the BusinessAgreementWithCoordinatorCompletion protocol where the *coordinator* tells the participants when the business activity is complete.

One major difference between this kind of protocols and those that deal with atomic transactions, is the fact that participating services are not required to remain participants for the duration of the activity. Because there is no tight control over the changes performed by services, they may leave the business activity after completing their individual contributions. When doing so, participants enter an exit state by issuing an exit notification message to the business activity *coordinator*. This feature ensures that service autonomy and statelessness are preserved since services are obliged to participate within an activity for only the duration they are absolutely required to.

WS-Composite Application Framework (WS-CAF)

Alternatively to the previously described specifications, WS-Composite Application Framework (WS-CAF) [Cover Pages WS-CAF; Wiki WS-CAF] is another specification for service coordination, which was firstly defined by a committee composed by Arjuna Technologies, Fujitsu Software, IONA, Oracle and Sun Microsystems, and then transferred to an OASIS Technical Committee.

WS-CAF shares the same goals with WS-Coordination and former WS-Transaction, having both an architecture and operation similar to WS-Coordination, comprising three distinct specifications:

WS-Context (WS-CTX) which defines a lightweight framework for simple *context* management that eases the share of a common *context* and propagation of a common outcome among all the Web Services involved in an activity. It is quite similar to the *Activation Service* of WS-Coordination but defines additional message exchanges that allow to query the content of a *context* or the state of coordination.

WS-Coordination Framework (WS-CF) is very similar to WS-Coordination without the *Activation Service*, as it provides a framework to spread the common *context* and results to all the Web Services that intervene in an activity.

WS-Transaction Management (WS-TXM) supports multiple transaction models to help enable participants to negotiate outcomes with each other and make a common decision about how to behave, specially in the event of a failure, whether the execution environment is CORBA, Enterprise JavaBeans (EJB), .NET, Java Message Service (JMS), or some combination of technologies. The WS-TXM encompasses three distinct, interoperable transaction protocols that can be used across multiple transaction managers:

ACID Transaction (TX-ACID) defines an activity that is bound to the scope of the transaction, so that its end triggers the termination of the associated transaction. It is compatible with existing transaction processing systems. Corresponds to the Atom transaction type in BTP and to WS-AT in the WS-Transaction family.

Long Running Activity (TX-LRA) defines activities that are long in duration, as well as the conditions and triggers for *compensation* actions to be activated if the activity is not successful. Allows to combine in the same activity both compensable and non-compensable services. Corresponds to the Cohesion transaction type in BTP and to WS-BA in the WS-Transaction family.

Business Process (TX-BP) has no direct correspondence both in BTP as well as in the WS-Transaction family due to its completely different nature. Its objective is to link or tie heterogeneous transaction domains into a *B2B* transaction. Each domain can use a different transaction model and can also save checkpoints to roll back any transient changes when failures occur.

These specifications were accepted as input by the WS-CAF Technical Committee, in the same state they were published by the referred multi-company committee on July 28, 2003. There is some parallelism among the three latterly described OASIS standards for coordination of services, both in the number and roles of the entities, and in the supported types of transactions. However, the standards, with the exception of WS-CTX whose specification for version 1.0 was published in April 2007 by OASIS, are nowhere to be found, having disappeared almost every referencing websites.

2.3 Web Services for fault tolerance

Reliability is a critical issue for service orientation [Khoshafian, 2007], as all the involved entities must be assured that all the message exchanges are reliable, and it may very well be the most important requirement of the *Quality of Service* (*QoS*). For instance, service security, transactional exchanges as well as business process management need reliable message exchanges.

Since the early 1990s, *Reliable Messaging* has been seen as a solution for such scenarios by the IT community, and thus, several message queueing technologies have been used, such as IBM's WebSphereMQ and Microsoft's MSMQ, in addition to reliable publish/subscribe technologies, such as Tibco Rendezvous. In an effort to bridge all these different technologies, the Java Message Service (JMS) was developed by the Java Community Process. Some of these technologies were adapted to Web Services but, due to the exploitation of proprietary protocols, interoperability can only be achieved recurring to gateways that mediate specific pairs of environments.

With the emergence of Web services as the preferred integration solution for distributed systems, it is now realistic to think about the possibility of a unified interoperability standard for *Reliable Messaging* [Weerawarana et al., 2005].

QoS includes advanced aspects of runtime processing such as reliability guarantees or service coordination among others [Margolis and Sharpe, 2007; Weerawarana et al., 2005]. Therefore, fault tolerance in Web Services may be included in the layer of protocols related to *QoS*, and it can be promoted through *Reliable Messaging* and transactions [Osrael et al., 2007a; Weerawarana et al., 2005]. There are many specifications for Web Services, but only few of them address the dependability of services.

When it comes to *Reliable Messaging*, there are two competing Web Services standards: WS-Reliability and WS-ReliableMessaging. WS-Reliability is older than WS-ReliableMessaging, but both provide message and sequence delivery assurances [Osrael et al., 2007a].

A Web Service can also support reliability through mirroring or replication where there are several entities to handle client requests to a certain service even in the event that some of these entities fail [Salas et al., 2006]. This scenario transposes the use of replication to service-oriented architectures. There have been many Web Services replication framework proposals, from active replication [Salas et al., 2006], to passive replication [Jayasinghe, 2005; Liang et al., 2003; Osrael et al., 2007b], passing by the Byzantine Fault Tolerance (BFT) [Lamport et al., 1982] technique [Li et al., 2005; Merideth et al., 2005; Pallemulle and Goldman, 2008; Zhao, 2007] and the N-Version

model [Looker et al., 2005]. However none has been standardized or widely used, as well as no replication standard has emerged. One of the reasons for this situation is the difficulty to apply replication across heterogeneous entities and domains [Osrael et al., 2007a].

Albeit the benefits they can bring in large scale and heterogeneous service oriented environments [Osrael et al., 2007a], standards for Web Services that provide other types of dependability mechanisms, such as failure detection, membership monitoring, or *Reliable Multicast*, still have not been proposed.

2.3.1 Reliable communication

Some *Reliable Messaging* standards for Web Services were defined in order to harmonize the use of already known techniques from operating and middleware systems to achieve that kind of reliability [Ferguson et al., 2003]. *Reliable Messaging*, simply put, means that a message that is sent must be delivered. But it can also include some other communication features:

- non-duplicate message delivery assurance;
- message sequencing assurance;
- non-repudiation help through sufficient bookkeeping.

Focus should be upon *Reliable Messaging*, since services are all about message exchanges over platforms and servers, which should achieve end-to-end reliability for the services to be usable.

However, there is another kind of reliable communication that should be object of further research, which is *Reliable Multicast*. Since many distributed algorithms for fault tolerance depend on *Reliable Multicast*, the standardization of multicast protocols for Web Services would be highly beneficial [Osrael et al., 2007a]. One recently proposed protocol is the SOAP-based WS-Multicast toolkit, that exposes its operations via WSDL [Salas et al., 2006].

WS-ReliableMessaging

WS-ReliableMessaging is an OASIS standard and its latest version, 1.2 [WS-RM], was approved on February 2, 2009, and it defines a protocol that allows messages to be delivered reliably between distributed applications in the presence of software component, system and network failures.

Several proprietary message-oriented middleware solutions that reliably route and distribute messages are used in the industry. WS-ReliableMessaging

allows to bridge two different infrastructures, such as different operating and middleware systems, into an end-to-end model where messages are exchanged reliably [Ferguson et al., 2003]. So, this standard ensures the interoperability of services in what comes to *Reliable Messaging*, which also simplifies the development of services, since they must implement the protocols, minimizing the number of errors in business logic [Ferguson et al., 2003].

The WS-ReliableMessaging specification distinguishes all the parts involved in an interaction, as well as the various meanings of the terms *send*, *transmit*, *receive* and *deliver*, as they relate to different components. In that sense, the basic model of WS-ReliableMessaging includes four distinct entities:

Application source Service or application logic that *sends* the message to the RM source;

RM source Physical processor or node that performs the actual wire transmission;

RM destination Target processor or node that *receives* the message and then *delivers* it to the application destination;

Application destination Target service of the message.

These nodes are endpoints, which according to the WS-ReliableMessaging standard, represent addressable entities that send and receive Web services messages.

To simplify, the basic mechanism of the standard works as follows: the source node sends a Web Service message containing a WS-ReliableMessaging header, which is received by the destination node that then replies by sending an acknowledgment message to the source node.

There are several types of assurances defined in the WS-ReliableMessaging standard, when it comes to message delivery:

AtMostOnce A message is delivered at most once, but it may not be delivered at all;

AtLeastOnce A message is delivered at least once, but it can be delivered more times;

ExactlyOnce This type is a combination of the previous two. A message is delivered only once;

InOrder When there are several ordered messages, they are delivered in the same order as they were sent.

WS-ReliableMessaging is a suitable standard to ensure point-to-point reliable message delivery. However, it would be very inefficient and introduce a heavy burden on the message sender in terms of processing power, if there are lots of message recipients or if many errors occur.

WS-ReliableMessaging can be used in conjunction with other WS-* standards, such as WS-AtomicTransaction, WS-BusinessActivity, WS-Coordination, WS-Addressing, WS-Security and WS-Policy. The WS-Security specification can be used to protect the sequences of messages that are governed by WS-ReliableMessaging. The WS-Policy framework can specify delivery assurances for sequences of messages defined by WS-ReliableMessaging, and it also enables a source and a destination of a *Reliable Messaging* interaction to describe their requirements [Erl, 2004, 2005; Juric et al., 2007].

WS-Addressing provides a mechanism to identify the address of a generic Web Service that is meant to receive replies or fault responses in the event of some problem occurring while using WS-ReliableMessaging [Juric et al., 2007]. WS-Addressing was modified to accommodate some needs of the WS-ReliableMessaging specification, like the reuse of a message ID when retransmitting identical messages to counter communication errors [Erl, 2005].

Although the WS-ReliableMessaging specification allows to condition service activities, it is different from WS-AtomicTransaction or WS-BusinessActivity, in the sense that a coordinating entity is not needed to inspect the progress of the activities, being the reliability rules conveyed as SOAP headers in the exchanged messages [Erl, 2005]. However, in order that WS-ReliableMessaging guarantees atomic delivery to all targets, it would have to rely on WS-AtomicTransaction, or a similar coordination protocol, which would increase the consumption of the sender's processing and communication resources, due to the additional message traffic. Hence, WS-ReliableMessaging, by itself, is not capable of dealing with failures in a cluster that provides a replicated service.

WS-Reliability

WS-Reliability is an alternative standard to WS-ReliableMessaging to provide the guaranteed delivery of messages on Web Services. Version 1.1 [WS-R] of WS-Reliability was declared an OASIS standard on November 15, 2004, and it provides the schema that can be used for conforming messages in reliability header blocks, and also an HTTP binding specification and fault handling messages. Just like in WS-ReliableMessaging, the basic model of WS-Reliability comprises two similar steps:

1. the source node sends a message with a WS-Reliability header block;

2. the destination node receives this message and then sends an acknowledgment message to the source node.

The SOAP header of the reliable message specifies its source, destination, ID, timestamp and a request for acknowledgment. The acknowledgment message will indicate the reliable message ID, timestamp, source and destination.

WS-Reliability also supports *QoS* message exchange patterns, such as duplicate elimination or guaranteed delivery, where reliable delivery of sequences of messages can also be included, just as WS-ReliableMessaging does.

2.3.2 Service replication

Replication is one of the primary fault tolerance techniques that has matured in some traditional fields such as databases. However, its use in service oriented environments is taking the first steps in terms of proven effectiveness and usefulness to achieve really dependable solutions [Osrael et al., 2007b].

In Service-Oriented Architectures, we can think of replication of two different layers of the architecture: the data layer (*e.g.* database back end) and the service layer (*e.g.* Web Services front end) on top of it. Replication can be applied to just one of this layers, or to both of them. On the data layer, it can be achieved through traditional database replication techniques provided by commercial or open source database management systems. If the Web Services on the front-end are stateful, which goes against the principle of statelessness, and the state information is stored completely on the data layer, replication still can be made through traditional database replication mechanisms. On the other hand, if the state is maintained in a data store where replication is impossible, state synchronization must be provided by some other means.

Few middleware solutions for Web Services were presented until now, and only some of them can be considered fault-tolerant and state-of-the-art in terms of the used Web Services technology, such as, WS-Replication [Salas et al., 2006].

The use of replication in critical scenarios is an intra-enterprise concern despite the intrinsic inter-enterprise nature of Web Services. So a replication middleware platform can be optimized for the technology used in each organization [Osrael et al., 2007a].

Service replication middleware can reuse many of the concepts used in distributed objects or replicated databases systems. However, the differences in the essence of the referred systems and Web Services, for instance their

coarse grained nature, allow for some optimizations in the service replication middleware.

The future of replication in ultra-large scale systems will include additional research to enable replication in a truly service-oriented way, regarding the heterogeneity of the building blocks of the systems [Osrael et al., 2007b].

A replication middleware for Web Services built upon the Java-based Axis2 SOAP engine is presented in [Osrael et al., 2007b]. The replication model of this middleware is a variant of the primary/backup, or *passive*, approach but also allows for behavior modifications through plugins with other replication protocols. Being more specific, this middleware forwards the client invocations to the backups, which have to process them, just like in *active* replication. This results in coherent state among all replicas if no non-deterministic operation is to be executed. However, this variant is different from *active* replication because it requires a multicast primitive (FIFO ordering) that is weaker than *Total Order Multicast*. The failover mechanism for this replication middleware resorts to consensus to determine the new master, or even to define the ordering of received invocations. This replication middleware assumes failures in nodes, which may crash and stop, and in network links, which may fail by losing messages only, not considering duplicate or corrupt messages.

Passive replication is also used in FAWS [Jayasinghe, 2005] and FT-SOAP [Liang et al., 2003], but without the modularity of the previous framework [Osrael et al., 2007b], which enables the usage of coordinator-cohort replication, for instance.

WS-Replication [Salas et al., 2006] offers transparent active replication and it relies on *Reliable Multicast*, supplied by WS-Multicast, which enables SOAP-based group communication and node failure detection via a SOAP-based ping mechanism.

WS-Multicast can also be used as a standalone component to enable *Reliable Multicast* in a Web Service environment. The proposed framework allows the deployment of a Web Service in a set of sites to increase its availability, and transparently forwards a normal web service invocation to its replicas using multicast. The reply to this invocation is sent back to the client when the service receives the configured number of responses, which could be one, a majority or all.

Byzantine Fault Tolerance (BFT) [Lamport et al., 1982] is a replication technique designed to protect against arbitrary problems like crash faults, software bugs or security violations and requires a higher degree of replication than crash faults tolerant techniques [Merideth et al., 2005]. The usage of a Byzantine-fault-tolerant service in Web Service applications implies that it must be able to attend non-replicated clients and interact with non-replicated

Web Services. The libraries included in the system deal with the underlying complexity of the BFT protocol and bridge communication with standard SOAP engines.

Thema [Merideth et al., 2005] provides a structured way to build Byzantine-fault-tolerant and survivable Web Services that are externally visible and accessible as standard Web Services. Thema incorporates the Castro-Liskov Practical Byzantine Fault Tolerance (CLBFT) [Castro and Liskov, 1999] protocol, in order to achieve a reliable and secure transport layer without any synchrony assumptions for safety. This middleware system provides SOAP and WSDL support for BFT, as well as adding multi-tier support to BFT, while working in a mixed-fault model.

BFT-WS [Zhao, 2007] is a BFT middleware framework for Web Services, and, like Thema, it is based on CLBFT. However, it builds upon WS-ReliableMessaging to achieve reliable control communication, using the regular SOAP/HTTP transport, in contrast with Thema which uses a wrapper to interface with a BFT protocol that relies on IP multicast, possibly introducing interoperability problems.

Perpetual-WS [Pallemulle and Goldman, 2008] also builds upon CLBFT and it attempts to address some of the shortcomings of Thema and BFT-WS, for instance, the support of replicated clients, or when a replicated Web Service has been compromised, *i.e.*, it has more than f faulty instances. Perpetual-WS supports long-running operations, as well as non-deterministic operations, such as local clock queries, pseudo-random numbers and timestamps, as the replicated Web Service will reach a consensus on the response to send back to the clients. Albeit these advantages, the latency introduced by this middleware almost doubles, as the BFT algorithm is both run on the Web Service replicas as well as on the replicated clients, to reach consensus on the received responses, in order to avoid that different responses are accepted by them. However, if the responding Web Service has been compromised, all the clients might receive the same malicious answer and still agree to accept it, which really seems not to solve the indicated problem, contrarily to what was promised in [Pallemulle and Goldman, 2008].

Similarly to Perpetual-WS, SWS [Li et al., 2005] also enables the interaction between Web Services with different degrees of replication, and it additionally supports dynamic discovery by adding the replicas endpoints information to the service's WSDL, to allow UDDI registries to store and serve information on Web Services and their replicas. However, it shares some of the shortcomings of Thema and BFT-WS.

A BFT algorithm that requires message ordering per source only, instead of total ordering, thus reducing inter-replica communication, was proposed in [Chai et al., 2013] to achieve trustworthy coordination of WS-BusinessAc-

tivity, allowing activities to tolerate the faulty behavior of the intervening parties. Albeit having better performance than other BFT protocols, this algorithm has limited application since it has been customized specially for WS-BusinessActivity, depending very closely on its state model.

WS-FTM [Looker et al., 2005] applies the N-Version model to Web Services in order to increase the dependability of a service, allowing it to tolerate both Byzantine and physical failures. It uses a simple-majority voting scheme for achieving consensus on the response to a client's invocation, by analyzing the replies sent by the equivalent N-Version services in response to the replicated invocation.

The Web Service Management System (WSMS) [He, 2004] is a comprehensive platform for the development, management and execution of Web Services, and it reacts flexibly to failures in order to ensure the correct and reliable functioning of services. It is capable of recovering from *Fail-stop* crashes, leaving Byzantine failures out of the equation, by using a heartbeat mechanism to detect failed services, and by replacing them by their corresponding backups.

2.3.3 Membership management

Membership management is one of the building blocks of fault tolerant distributed systems, as it allows to keep track of the components comprising such systems and their current state, enabling the detection of failed components. WS-Membership, which was built on top of WS-Coordination, proposed a framework that provides cooperating Web Services and activity monitors with a unified approach for tracking registered Web Services and for supplying membership updates to monitors [Vogels and Re, 2003]. However, it was not standardized and seems to have ceased to exist as little information can be found on the internet. This protocol was developed in the context of the *Obduro Project*, which aimed to apply the results of scalability and reliability research to global scalable Service-Oriented Architectures, and also seems to have been terminated. In WS-Membership there are five different roles defined:

Coordination Service Receives activation and registration requests, which are then routed to the Membership Service.

Membership Service Provides failure detection of registered Web Services and propagates membership information.

Member Service Registers with a Membership Service, or through a Membership Proxy, for failure detection.

Membership Proxy Software component interposed between a Member Service and the Membership Service for reasons of efficiency or accuracy.

Membership Monitor Registers with Membership Service its interest in receiving membership state updates.

The Membership Service is based on epidemic techniques which are adequate to achieve loosely coupled, asynchronous, autonomous and distributed components. Since gossip-style communication is used, the exchange of membership information is highly robust and occurs asynchronously. These reliability and scalability properties, among others, enable this protocol to be used in large federated environments. Albeit the disadvantages of using epidemic failure detection, like inefficiency when the size of messages grows proportionally with the number of participants and bad behavior with massive concurrent participant failures, the detection of failed Member Services is very accurate.

A standard for failure detection and membership monitoring would be highly beneficial due to the heterogeneity of the systems used in Web Services based environments. Failure detection can also be used as the building block to simplify the implementation of other essential distributed systems services such as consensus [Vogels and Re, 2003].

2.4 Gossip protocols

In peer-to-peer computer networking, gossiping describes the process where a participant that intends to disseminate some information randomly chooses a small subset of other participants and forwards that information to them. Each of these destinations, upon receiving the information, repeats the same procedure, hence, the gossip moniker. This also mimics how epidemics spread in populations, justifying the alternative denomination of epidemic protocols [Eugster et al., 2004], but instead of spreading a virus, information is transmitted from one node of the network to other randomly chosen nodes.

Epidemic algorithms have received much attention recently, mostly due to their intrinsic robustness, simplicity and scalability [Eugster et al., 2004; Karp et al., 2000; Kermarrec and van Steen, 2007a] among many other reasons, such as speed and persistence of dissemination, ease of deploy and fault-tolerance.

This section reports the origins and basics of epidemic algorithms, followed by their application to communications as gossip protocols, and ending with the relationship between the latter and overlay networks.

2.4.1 Background

The first known use of epidemic algorithms was for the update and synchronization of data among the many sites of a replicated database [Demers et al., 1987], by using two different epidemic strategies:

Anti-entropy Each site randomly selects another site, at regular intervals, to synchronize their states by exchanging the whole database.

Rumor mongering Upon reception of a new update, a site randomly selects another site and sends the update to it, until some termination criterium is fulfilled.

Anti-entropy is extremely reliable whilst producing a large overhead on communications, reason why it should not be used very frequently. Rumor mongering generates significantly less overhead on communications because only recent updates are exchanged. The best combination of these two strategies would be to use rumor mongering frequently whilst anti-entropy should be used very rarely [Karp et al., 2000].

Rumor mongering comprehends two different kinds of transmission: *push* and *pull* [Demers et al., 1987]. *Push* transmission was the original idea for rumor spreading and many termination mechanisms, which determine when a node should stop transmitting an update, were investigated. *Pull* transmission specifies that the targeted node will send the rumor to the calling node. This strategy has proved to be more performant than the previous one, since the number of rumor-ignorant nodes decreases much faster than when using *push* transmission, and should be used when updates occur frequently.

More recently, gossip has been used for building a bimodal multicast protocol [Birman et al., 1999], whose model addresses failures of processes and communication, contrarily to previous work, which just considered communication failures.

2.4.2 Epidemic algorithms

In an epidemic algorithm, all the processes that make part of a system are potential disseminators of messages. Every time a node receives a new message it becomes a sender, except in the case of duplicate messages, as happens with an infected person that can not be infected again.

Parameters

To define the behavior of the processes there are several dissemination parameters that can be defined:

Buffer capacity (b) Maximum number of messages that a process buffers and then resends;

Rounds (r) Number of times that a message is going to be forwarded by some process;

Fanout of the dissemination (f) Size of a set of randomly selected processes that will receive the message forwarded by the current process.

There are many variants of epidemic dissemination algorithms that can be distinguished by different values attributed to the latter parameters [Eugster et al., 2004]. An additional parameter is n that corresponds to the number of processes in the system. The reliability of the message delivery depends on the values of all these parameters.

The reliability of these algorithms is based on a *proactive* mechanism where redundancy and randomization are used to avoid potential process and network link failures. In this mechanism, every process chooses randomly a subset of size f of the remaining processes to which the message is then forwarded. Each of these processes behaves exactly in the same way when it receives a message, so there is no *reactive* mechanism to deal with failures. However, processes that fail permanently have to be removed from the system.

Issues

The implementation of epidemic dissemination algorithms raises several questions. Some solutions [Eugster et al., 2004] have been defined for the following identified issues:

Membership The way processes know a necessary minimum amount of their neighbors;

Network awareness How processes become aware of the current network topology in order to avoid broken links;

Buffer management The algorithm that decides which message or messages should be dropped when the buffer of a process fills up;

Message filtering How to reflect the interest of processes in some information so the probability that they receive and store information of no interest to them is decreased.

Studies of natural epidemics can provide useful information regarding these issues. However, since their primary target is to terminate the contagion, a different approach should be pursued to study how to ease the information dissemination.

Variants

There are several variants of gossiping [Karp et al., 2000; Pereira et al., 2006], which provide different message exchange patterns and performance trade-offs, and should be used according to several conditions, such as message size and network bandwidth.

The two main gossip variants, as mentioned previously, are:

Push A node that knows of new information, conveys it to another node. This process is repeated for a specified number of times. This variant is adequate for one-to-many dissemination of small messages and events.

Pull Instead of gossiping upon arrival of new information, a node periodically selects a number of peers and asks them for new information, and receives a response if that situation is true.

It has been shown that combining *push* and *pull gossip* results in dissemination being achieved in a lower number of steps [Karp et al., 2000] and provides a generic framework for gossiping that can be tailored for multiple purposes by parameterizing it with different aggregation functions [Jelasity et al., 2003]. In addition, lazily deferring the transmission of payload improves performance in heterogeneous networks, allowing gossip protocols to approximate ideal resource usage efficiency [Pereira et al., 2006]. The following variants are the *lazy* versions of both *push* and *pull* variants:

Lazy push A node that knows of some new data sends only the corresponding information topic. An interested receiving node contacts another node and, by sending the information topic, identifies the desired data. If the contacted node already has it, it just passes it through to the interested node. Otherwise, it forwards the request that eventually reaches the originator of that data, that will then complete the information transmission.

Two-phase pull Very similar to *pull* but where the target node sends only the recent information topic which must be asked for, explicitly by an interested node.

Both these lazy variants are useful when the data payload is very large, but *lazy push* is also useful when it is very likely that the data is already known throughout the network.

Another variant of gossiping, *Hybrid Push Gossip*, which combines the two push strategies in an epidemic multicast protocol to achieve better performance in heterogeneous networks is presented in [Pereira et al., 2006]. In this variant, *eager* or *lazy push* is selected independently for each target to whom the message will be sent at each round.

Two different models of gossip, which differ in the behavior of the infected nodes when dealing with duplicate messages, are referred in [Eugster et al., 2004]. In the *infect-and-die* model, a node that is infected, *i.e.* receives a message, takes only one round to send the received message to other nodes, and then never sends it again, becoming dead in the analogy with nature. In the *infect-forever* model, also known as 'Balls and Bins' [Koldehofe, 2002], a node does not die, which means it can send a received message multiple times, possibly until r rounds are reached or some other stoppage criterium is satisfied. This last alternative has the advantage of requiring no state at participants to recall recently relayed messages. On the other hand, it usually requires more network resources as the relay limit has to be set conservatively.

Reliability

Epidemic multicast protocols disseminate data efficiently among a large number of nodes while providing a probabilistic guarantee of delivery, throughput stability as well as high resilience in the event of node or network failures [Pereira et al., 2003].

Gossip configuration parameters and fault probability determine the mathematical formulas that characterize the reliability of epidemic information dissemination and allow the algorithm to achieve high reliability even if processes crash or disconnect, packets are lost or even if the network topology is highly dynamic [Eugster et al., 2004].

The *lazy* variants are more prone to network faults, due to the additional round-trip, which also increases latency [Pereira et al., 2006].

Most interestingly, gossip protocols do not need a reactive mechanism to deal with failures, namely, buffering, acknowledgement, retransmission, and garbage collection, which account for most of the complexity in common communication protocols. Instead, reliability is proactively achieved by the protocol's inherent redundancy and randomization, that cope with both process and network link failures.

The expected probability for a message being delivered to each destination and to all destinations as a whole can be derived directly from protocol

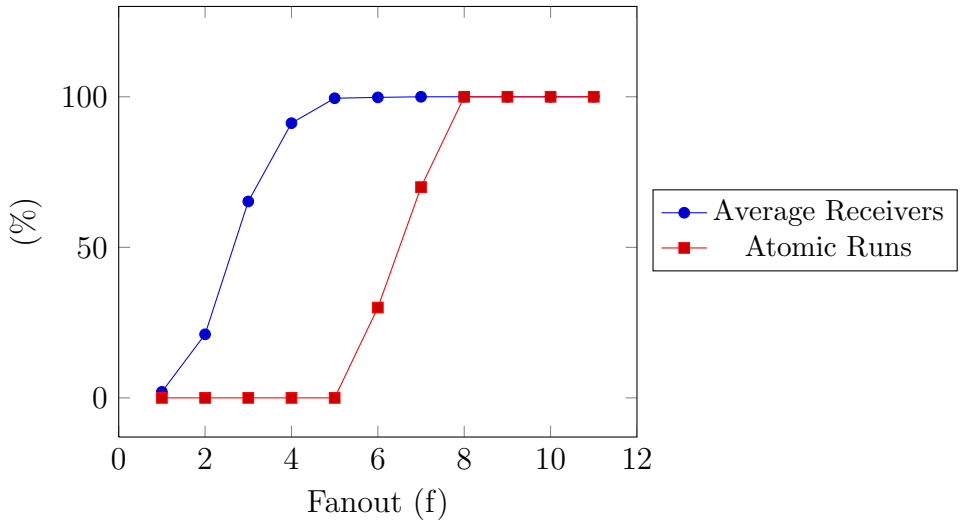


Figure 2.3: Reliability of gossip (250 participants, 10 dissemination runs, variable fanout).

parameters f, the number of targets that are locally selected by each process for gossiping, and r, maximum number of times a message is relayed before being ignored. Figure 2.3 illustrates the impact of these parameters by showing simulation results of disseminating 10 messages to 250 receivers, with $r = 5$ and a variable f. Notice that with $f > 4$ each destination gets each message with a very high probability. With $f > 7$, each message is atomically received by all destinations also with a very high probability.

By adjusting r and f parameters according to system size and expected faults, gossip can be configured such that any desired average number of receivers successfully get the message. Better yet, parameters can be set such that the message is atomically delivered to all the receivers with high probability leading to guaranteed atomic delivery [Eugster et al., 2004]. The key to scalability is that the required fanout configuration is at worst logarithmically proportional to system size.

Performance

Message dissemination speed increases exponentially and the probability that all the nodes in a network have received it, can be very close to 1, through tuning of configuration parameters, however without ever reaching that value [Eugster et al., 2004].

The *lazy* variants of gossip are useful because the probability of sending a message several times to a node is decreased, thus saving network bandwidth

and processing power of the nodes [Pereira et al., 2006]. In the first rounds, the *lazy push* variant is not so performant since only a small part of the nodes has received the message and its transmission will certainly be requested. *Eager* variants are not so performant in the last rounds as they introduce a large overhead. This is due to the fact that nodes start receiving multiple copies of a message, since a large amount of the nodes have already received it.

To get the best out of the various types of gossip, it is advised to switch from *eager* to *lazy* gossiping based on the round number [Pereira et al., 2006], as exploited in the *pbcast* protocol [Birman et al., 1999], or to use an *eager push* strategy initially and then switch to a *pull* mechanism [Karp et al., 2000].

2.4.3 Membership management

A key component of a gossip protocol is the ability to obtain random subsets of participants to direct messages at in each gossip operation. This component has to provide an uniform random sample and, as much as possible, drawn from a current view of operational participants [Jelasity et al., 2004]. The first option is to share the full list of participants, allowing each of them to locally draw subsets as desired [Birman et al., 1999]. This is adequate when the list does not change frequently, to avoid taxing the network with constant updates, and is small enough to fit each participant's memory.

If these conditions are not met, it has also been shown that sufficiently good random samples can be obtained by having each participant keep a small partial view of the system, which is itself maintained using a gossip protocol [Eugster et al., 2003, 2004]. A particularly simple but effective approach [Voulgaris et al., 2005] is allowing a node to exchange some elements in its local list with the same number of elements from some other node. This progressively shuffles the list of each participant and leads to the desired uniform random sample. By adding a time-based lease and renewal mechanism, it also deals with participants entering and leaving the system.

2.4.4 Overlay networks

Overlay networks correspond to a type of computer networks that build upon other networks. They can be seen as logical networks, built on physical networks, which provide communication links that allow communication to take place among the nodes of those networks [Wiki Overlay network].

The membership issue of gossip protocols is directly related to overlay networks maintenance, since a node must, somehow, obtain updated infor-

mation on its neighbors for message exchanges to be performed. Turning this relationship the other way around, the construction and maintenance of overlay networks can be performed resorting to gossip protocols [Jelasity et al., 2004].

A peer sampling service can be used to provide every node with the list of the peers that were selected for it to exchange information with. The invocation of this service, at each iteration, by a node using a gossip protocol to communicate, would allow it to disseminate some information, or a request for it, to the nodes in the list provided by the peer sampling service [Jelasity et al., 2004].

Analytical studies have shown that gossip-based protocols can be highly reliable and efficient, but under the assumption that the used peer sampling service provides a list of peers selected uniformly at random from the entire set of peers. A more scalable and efficient implementation of such a service would be to gossip membership information in order to construct and maintain dynamic unstructured overlays [Jelasity et al., 2004].

A generic framework to implement peer sampling services using the latterly described strategy was presented in [Jelasity et al., 2004]. It allows to reproduce the behavior of existing approaches as well as new ones, and allowed to show that none of the different services performs an uniformly random selection of peers, which renders the traditional theoretical approaches invalid [Jelasity et al., 2004]. Instead, the different resulting topologies can be considered to belong to the family of small-world graphs, which are characterized by small diameter and large clustering.

Other conclusion withdrawn from the performed experiments with this generic framework is that from the different parameter settings result very different properties, which can be exploited to better fulfill the needs of the used application. For instance, a strong self-healing topology may not be suitable for scenarios where frequent network partitions occur. Possibly, the best combination can rely on the usage of a second view for gossiping membership information, and of several concurrently running protocols [Jelasity et al., 2004].

2.5 Consensus algorithms

Consensus is used when a set of processes need to agree upon the outcome of an operation. The consensus problem can be defined as the agreement on a set of processes to decide on a common value even if each of them starts with a different value, and regardless of the occurrence of faults. It corresponds to an abstraction of the problem of all processes in a fault-

tolerant distributed system agreeing on the same value despite having started with different opinions [Pereira and Oliveira, 2004]. The uniform version of consensus is defined by the following three properties [Guerraoui and Raynal, 2003; Guerraoui and Schiper, 2001]:

Uniform Agreement Every process decides on the same value.

Termination Every correct process eventually decides.

Uniform Validity If a process decides on a certain value, then it must be the initial value of some process.

The fact that there is no deterministic solution for the consensus problem, in an asynchronous system, even if a single process crashes, has been captured in [Fischer et al., 1985], and it is known as the FLP impossibility. Various solutions to this issue have been proposed, such as the usage of random or failure detector oracles [Chandra and Toueg, 1996; Chandra et al., 1996].

Many consensus protocols require knowledge regarding the processes involved in the execution of a protocol to establish a notion of majority, quorums, etc. Even though some protocols, such as Paxos, do not use failure detectors in their specification, agreement protocols usually rely on them or some similar functionality [Vogels and Re, 2003; Wiesmann et al., 2003]. In the case of Paxos, the Ω oracle is used to elect a new leader when needed. The Ω oracle is the weakest that can elect a new leader and the $\Diamond S$ oracle is the weakest that allows to solve consensus [Chandra and Toueg, 1996; Chandra et al., 1996]. It has been shown that Ω and $\Diamond S$ have the same computational power [Chandra et al., 1996].

Failure detection is usually done through the use of *heartbeat* or *are-you-alive* messages, but more advanced techniques exist [Wiesmann et al., 2003] such as the timeless failure detector proposed in [Mostefaoui et al., 2004].

Consensus protocols can be centralized or decentralized in the way the votes are collected [Pereira and Oliveira, 2004]. In a centralized protocol, a round coordinator collects the estimates from the previous round and broadcasts a selected estimate. Then, it collects all the votes and, finally, broadcasts the choice of the majority. This implies three communication steps to reach a decision. In a decentralized protocol, all the votes are broadcast to all the participants, allowing each of them to reach a decision by gathering a majority of the votes. This way, only two communication steps are required to reach a decision, but there is additional load put on the network.

2.5.1 Services based on consensus

Group communication system

The importance and usability of consensus protocols can be shown by its use as one of the basic components, or services, for the design of group communication systems [Wiesmann et al., 2003]. The implementation of these basic services should achieve four design goals in order to obtain a modular and structured architecture. The referred design goals should be:

Easy Substitution Easy exchange of the implementation of a service, that should provide the same functionality.

Autonomy When used separately, the service should retain its meaning and functionality.

Self-Containment Existing implementations of a service are easily integrated into a system.

Standard Services implement standard interfaces and protocols to provide some functionality.

Generic consensus service

Fault-tolerant agreement protocols can be built recurring to a generic consensus service [Guerraoui and Schiper, 2001]. To solve each different agreement problems, a different version is derived from a generic consensus filter in order to obtain an adequate protocol. Various agreement problems were reduced systematically to consensus, hence obtaining simple and original solutions [Guerraoui and Schiper, 2001]. The architecture of the proposed system can be divided into three layers:

Communication and Failure Detection Basilar layer of the system that allows processes to communication and to detect failures, which can be subdivided into:

- An asynchronous communication model, where channels are eventually reliable. Two communication primitives are used: *reliable multicast* (*Rmulticast*), which is stronger than *multisend*, because the latter can lead to partial reception of a message.
- A distributed oracle that can be accessed by processes via a local failure detector module. The failure detector satisfies the *strong completeness* property.

Generic Consensus Service Two different approaches were used for consensus implementation: a centralized one, where a coordinator takes the consensus decision; and a decentralized one, that does not use a coordinator.

Agreement Protocols Protocols used to solve problems related with distributed systems, such as non-blocking atomic commitment, group membership, view synchronous communication or total order multicast.

The framework presented in [Guerraoui and Schiper, 2001] has three different process roles:

Initiator The process that starts an agreement problem;

Client One of the processes that have to solve an agreement problem;

Server One of the processes that solve consensus.

These process roles can overlap completely, *i.e.* a process may be the *Initiator* of some agreement problem, and also *Client* and *Server* in the same problem, which might just be the typical scenario [Guerraoui and Schiper, 2001].

The interaction between *Initiator*, *Clients* and *Servers* relies on the *Rmulticast* and *multisend* communication primitives. The most basic interaction, corresponding to an agreement, as perceived by a *Client*, comprises three steps:

1. The *Initiator* multicasts a message to the *Clients* using *Rmulticast*.
2. *Clients* invoke the consensus service, *i.e.* contact the *Servers*, using the *multisend* primitive.
3. The consensus service sends the decision back to the *Clients*, also using the *multisend* primitive.

A consensus filter is attached to every *Server* process and contains a predicate *CallInitValue* that defines the necessary condition to activate the function *InitValue*, that also belongs to the filter. This function starts the consensus protocol.

The centralized approach uses the consensus algorithm presented in [Chandra and Toueg, 1996], which is identified as $\Diamond S$-consensus, and requires a majority of correct processes and a failure detector of class $\Diamond S$. An optimization for this algorithm was proposed in [Guerraoui and Schiper, 2001],

and states that it is sufficient the existence of one correct *Server* with an initial value when the consensus service is invoked, for the algorithm to work properly. This means *Clients* can send their messages to only one consensus *Server* but they must not suspect that it may have crashed.

The distributed approach has no coordinating process and takes advantage of the validity process of consensus. That is the same to say that, if all the consensus *Servers* start with the same initial value, it will be the *consensual* value. This scheme may take only three steps or message exchanges while the centralized scheme takes at least five steps. Another advantage of this scheme, compared to the centralized one, specifically in a network that allows broadcast communication, is that sending a message to various processes costs the same as sending the message to a single process.

To examine the benefits of using the proposed framework, the performance of the Non-Blocking Commit (NB-AC) protocol, built with the framework, and of the Three Phase Commit (3PC) protocols, proposed in [Skeen, 1981], were compared.

NB-AC has the advantage of higher modularity that allows to trade the number of exchanged messages against the communication resiliency and both the centralized and the decentralized schemes require less messages than 3PC and D3PC respectively. This confirms that the proposed consensus-based NB-AC protocol is more efficient than a 3PC protocol.

This framework can help to build practical systems that use different paradigms, and in that context, consensus is useful, not only as a theoretical concept, but also as a service for the clean development of reliable distributed systems.

2.5.2 Consensus based on services

Indulgent consensus

A generic framework for indulgent consensus [Guerraoui and Raynal, 2003] highlights the commonality in the design of various consensus protocols, and specifies the properties that oracles have to satisfy for the consensus protocols to terminate. This makes the framework modular, by requiring that only the used oracles satisfy the required properties, and also by allowing new oracles to be designed specifically for the environment of the systems where they are inserted. The fact that, in this framework, consensus is indulgent means that the resulting protocol never violates its safety properties even when the underlying oracle behaves arbitrarily and does not meet its specification.

The computational model defined in [Guerraoui and Raynal, 2003] consists on a finite set of processes, where they behave correctly until, possibly,

they crash due to some failure. The majority of the processes is correct, *i.e.*, they do not crash. Communication is established between processes through a reliable channel that conveys messages. The system is asynchronous because there is no assumption about the relative speed of processes or communication.

As a consensus protocol should match all known consensus lower and upper bounds, [Guerraoui and Raynal, 2003] focus on:

Power of the underlying oracle Has to be as weak as possible. The unreliable failure detector $\Diamond S$ [Chandra and Toueg, 1996; Chandra et al., 1996] is the weakest that allows to solve consensus.

Resiliency Has to be the highest possible and corresponds to the upper bound on the number of processes that can fail. Has to be less than half the number of processes for all indulgent protocols.

Latency in well-behaved runs Corresponds to the number of communication steps required to decide in a well-behaved run. A well-behaved run is one where there is no faulty process and the oracle behaves perfectly. Two communication steps is the minimal latency that can be achieved.

Zero degradation When the decision does not require more communication steps in stable runs than in well-behaved runs. A stable run is one where there are only initial crashes, as failures, and the oracle behaves perfectly.

Same initial values When this happens for all processes, no underlying oracle is needed to obtain a decision. In this case, a decision should be obtained on the minimal communication latency, that is two communication steps.

The protocol uses a *Reliable Broadcast* primitive to send messages reliably to the processes. The main concerns of this framework were generality, simplicity and also efficiency, so it should match all known consensus lower bounds.

The protocol framework proceeds by asynchronous consecutive rounds, which are composed of two phases each. On the first phase, it tries that all the processes have the same value, so it is possible to take advantage of the consensus termination property. The aim of the second phase is to guarantee that the consensus agreement property is not violated.

The use of the oracles encapsulated in the function *oracle*, has several advantages:

Modularity of the proofs For any instantiation of the function, it has to be proved that the required properties for the general protocol are satisfied.

Communication cost of a protocol instance The cost of the first phase depends on the used oracles. When using $\Diamond S$ or Ω [Chandra et al., 1996], it is one communication step, which provides optimal latency. When the Ω oracle is used, the resulting protocol enjoys the zero degradation property.

The *oracle* function must satisfy the same properties shown by the corresponding oracle that would allow consensus to be achieved. This oracle is, then, formally characterized by the following features: validity, quasi-agreement, fixed point, termination and eventual convergence. Modules correspond to implementations of the *oracle* function for the various types of oracle. It is also possible to combine modules in any way, provided the resulting combination still satisfies the same properties defined for the *oracle* function.

Three different types of oracles are considered:

Leader Oracle Must satisfy the **Eventual Leadership** property, which states that after some instant of time, every invocation to the oracle by correct processes will always return the same correct process that was defined as the leader.

Failure Detector Oracle Provides every process with a set of processes that are suspected to have crashed. It belongs to the $\Diamond S$ class if it satisfies the following properties:

Strong Completeness Eventually, every process that has crashed is permanently suspected by every correct process.

Eventual Weak Accuracy There is a time after which some correct process is never suspected by the correct processes.

Random Oracle Returns a randomly chosen value to the processes. For simplicity, only binary consensus was considered in this type of oracle.

A process starts a consensus execution by invoking the function *Consensus* passing the value it proposes as a parameter. The two phases that constitute each round of the execution of the algorithm, implement a 'two-phase commit' scheme: during the first phase, the processes try to select the same value, whereas, on the second phase, they try to decide, which occurs when they all have the same value at the end of the previous phase.

There are some optimizations for this framework [Guerraoui and Raynal, 2003], such as specific configurations where the decision can be obtained in one communication step. But to maintain the interest on the practicality of the framework, the used configurations should not condition the behavior in the presence of different initial settings.

Failures are rare in practice and the assumption that less than a third of the processes can crash allows to eliminate the need to use *reliable broadcast* and decide for a value if it receives more than twice the number of failed processes with that value.

Mutable consensus

A consensus protocol that can modify its behavior by balancing latency with the number of transmitted messages was proposed in [Pereira and Oliveira, 2004]. The selection of the behavior depends on the available computing power for processing messages, and network resources, where the cost of sending and transmitting messages is weighed.

The performed benchmarks on the new consensus protocol based on *stubborn channels* show that its performance is not appealing, but allows to extract an interesting property: as messages can be lost by *stubborn channels*, it is possible that only a small fraction of the messages sent by the protocol are actually transmitted through the underlying network [Pereira and Oliveira, 2004].

To explore this property, an implementation which maximizes the likelihood of desirable runs that exchange a small amount of messages at the network level, by introducing finite delays in a naive implementation of *stubborn channels* was developed [Pereira and Oliveira, 2004]. The correctness of the protocol, which assumes an asynchronous system model where processes only fail by crashing, is not compromised as the delays are finite.

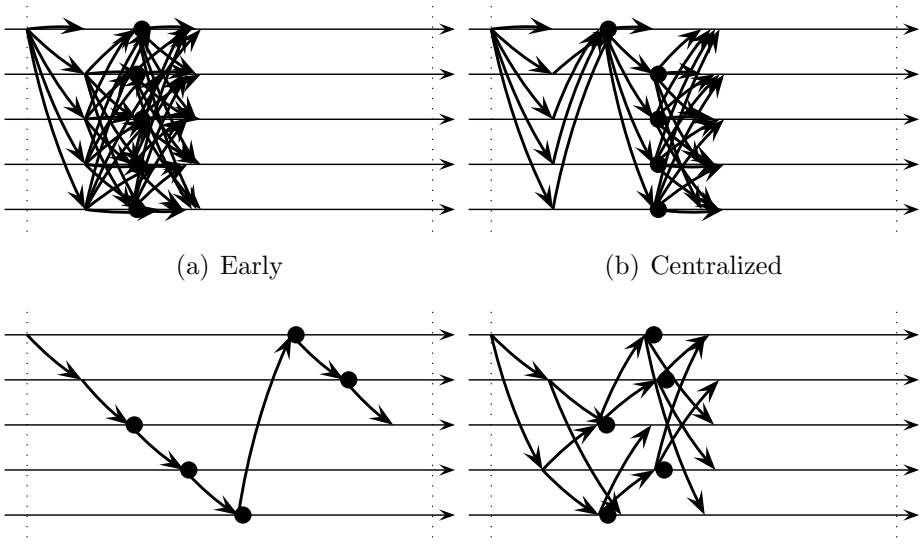
Very good performance can be achieved in practice when only desirable runs occur, as a result of carefully chosen delays. These delays avoid the actual transmission of a message m, possibly due to two different reasons [Pereira and Oliveira, 2004]:

1. They increase the likelihood of a more recent message being sent in the meantime, which discards all the previously sent messages.

2. If a decision can be reached by all processes before the delay expires, the transmission of m is avoided.

The various classes of desirable runs that are analyzed in the article result from different configurations of delays. Some of them resemble the message

exchange pattern of well known protocols, while others produce groundbreaking message exchange patterns with great performance features [Pereira and Oliveira, 2004]. For this reason, the protocol was named *mutable consensus*, and each combination of it with an implementation of *stubborn channels* is a *protocol mutation*.



(a) Early (b) Centralized

(c) Ring (d) Perm. gossip (F=2)

Figure 2.4: Prefixes of typical executions (1ms). Figure extracted from [Pereira and Oliveira, 2004].

Four different mutations, whose message exchange patterns can be observed on Figure 2.4, are studied in [Pereira and Oliveira, 2004]:

Early Simplest mutation which aims at a message exchange pattern where every process multicasts its vote to all the others every round. This pattern is similar to that of early consensus [Schiper, 1997] and allows the occurrence of decisions after two communication steps.

Centralized Has a message exchange pattern similar to the centralized algorithm of [Chandra and Toueg, 1996]. Compared to the previous mutation, it does not reproduce exactly the original protocol because the coordinator does not gather estimates when entering a round. It is very close to the *early* mutation, because it differs only by avoiding the direct transmission of votes among participants, which are then relayed by the coordinator.

Ring A ring structure is formed among the processes, where each of them communicates only with its successor.

Permutation Gossip Uses a gossip-style message exchange pattern with deterministic safety and liveness. Each process randomly generates a sequence of process identifiers, which is used as a circular list for the transmission of messages. A process sends a message to the first F processes on that list and the pointer is incremented by F. This value corresponds to the fanout value. Differently from the other mutations, the message delays vary according to each destination and they are computed using the same rule and regardless of the content of the message. The way a message is delayed ensures that, in at most n/F periods, it will have been transmitted to all processes.

The approach to obtain a mutable protocol, that allows reconfiguration to mimic protocols with different message exchange patterns is done in three steps [Pereira and Oliveira, 2004]:

- Build an algorithm that is correct in an asynchronous system model.
- Modify the algorithm to use *stubborn channels*, with the guarantee that it can skip messages that could have been discarded or duplicated.
- Ensure that the resulting algorithm provides a large number of different message exchange patterns.

Due to the unreliable nature of the communication channels, the reception of all messages by the processes is not ensured, which may lead to the exclusion of processes during some rounds. When a process receives a message from a larger round, it jumps directly to it.

A general impression on the behavior of each mutation can be extracted from Figure 2.4, but, by analyzing the charts in [Pereira and Oliveira, 2004], a better and deeper understanding can be achieved.

Both the *early* and *centralized* mutations do not scale regarding network and CPU usage. In the event that the latency of the communications is not critical, the *ring* mutation would be an excellent choice due to its extremely frugal usage of resources. The *permutation gossip* mutation is scalable to a large number of processes and still achieves low latency and resiliency to crashes and network failures.

Apart the results of these four mutations, there was another analyzed alternative, *random mix*, that corresponds to the situation where each process randomly selects which of these mutations to use in a consensus instance.

This use of the protocol demonstrates its correctness because there were no blocked processes, but also shows that the overall performance is poor.

There are two contributions in [Pereira and Oliveira, 2004]:

- A mutable consensus protocol which can be customized to use a certain message exchange pattern through a simple technique that preserves correctness and can be applied in run-time and without any coordination.
- The permutation gossip mutation allows the implementation of an efficient and scalable consensus protocol.

The proposed mutable consensus protocol is interesting due to the extracted abstract nature that is common to various consensus protocols. Since the performance tuning only affects the time domain, the correctness of the protocol, that assumes an asynchronous system model, is ensured. This tuning consists on the modification of the strategies used to compute the message delays. This selection of strategy can be done individually by each process, and is based on the semantics of the messages, as in [Pereira and Oliveira, 2004], or, for instance, network conditions or even other factors from the environment.

The *permutation gossip* mutation performs very well, outdoing other mutations in terms of scalability. It was also shown that the amount of messages to be handled by each process is more important performance-wise than the number of communication steps required for a decision, when the system has a large number of processes and limited resources. This fact proves the utility of being able to mutate the protocol according to the conditions of the system.

Protocol mutation is possible because it assumes lossy channels and, also, the received messages are always relayed. One proposition for future work was to develop mutable protocols for other distributed programming problems.

Paxos

The Paxos algorithm [Lamport, 1998] results from the application of consensus to the state machine approach. Consensus protocols are the basis for the state machine approach to distributed computing, as proposed in [Schneider, 1990]. This technique allows the conversion of an algorithm into a fault-tolerant and distributed implementation, through the ordering of all the actions involved. This ordering mechanism depends on the synchronization of the actions among all the processes or nodes involved in the distributed

system. To achieve consistent order in all the nodes, a consensus protocol is essential. This approach also handles safely all cases of failure, since failure can only be perceived in the context of physical time, by a user or a process if a supposedly failed process is taking too long to respond.

The Paxos designation defines a family of protocols to solve consensus in a network of unreliable processors, and it includes a spectrum of tradeoffs between the number of processors, number of message delays before learning the agreed value, the activity level of individual participants, number of messages sent and types of failures. The common property to all of them is their safety from inconsistency.

In [Lamport, 2001], an asynchronous and non-Byzantine model is assumed, where processes, that operate at an arbitrary speed and may fail by stopping and then restart, communicate through the exchange of messages that are not corrupted, but may take long to be delivered, duplicated or lost.

There are five different roles in the Paxos protocol [Wiki Paxos], and a single processor may perform one or more roles at the same time. All of the roles are described next:

Client Issues a request to the distributed system, and waits for a response.

Acceptor *Acceptors* are grouped into Quorums, that correspond to a simple majority set of the existing *acceptors* in the system. Every message that is sent to an *acceptor* must be sent to a Quorum. Any message received from an *acceptor* is ignored unless copies are received from all the *acceptors* in a Quorum.

Proposer A *proposer* advocates the request of a *client*, for the *acceptors* to agree on it, and acts as a coordinator to ensure the progress of the protocol when conflicts arise. A *proposer* can also make multiple proposals, as long as it follows the algorithm for each one, and can abandon a proposal at any time.

Learner A process with the role of *learner* aims to know about values that are chosen. Once the request of a *client* has been agreed on by the *acceptors*, the *learner* may perform the corresponding action. A special *learner* role is determined, the distinguished *learner*, that receives the acceptances of the *acceptors*. In order to augment reliability, this role should be performed by several processes, which then inform other *learners*. Because of message loss, a *learner* may never know about a chosen value, and the best way to avoid this, is for it to have a *proposer* to issue a proposal of that value.

Leader In Paxos, a leader is selected to perform the roles of distinguished *learner*, and possibly also of distinguished *proposer*. In the event that various processes believe to be *leaders*, one of them must be chosen eventually for the protocol to progress. The safety properties are preserved even if the protocol stalls due to the existence of two concurrent *leaders*.

Additionally to these roles, there are two more concepts in Paxos worth describing:

Quorum Expresses the safety properties of Paxos by ensuring that some surviving processor retains knowledge of the results. Corresponds to any majority of participating *acceptors*.

Choice Sometimes, the *leader* has to choose among conflicting values, and the selection is not determined by the protocol. So, the choice must be one of the values from the most recent round, and, typically, the majority value from the highest round is chosen. Whatever the choice may be, correctness is guaranteed.

A *proposer* sends a value to a set of *acceptors*, and consensus is said to occur when a majority of the *acceptors* have accepted it. This scheme works if an *acceptor* accepts at most one value, because any two majorities of *acceptors* have at least one common element which should not change its accepted value.

Acceptors can always respond to *prepare* requests and accept a proposal numbered n if it has not responded to another *prepare* request numbered higher than n. But an *acceptor* can ignore any *prepare* or *accept* requests without compromising safety. An *acceptor* can ignore a received *prepare* request numbered n, if it has already responded to a *prepare* request numbered with the same value n or greater than n. This way, an *acceptor* can only remember the highest-numbered proposal it accepted and the number of the highest-numbered *prepare* request it has responded to.

If a sufficient part of the system is working properly, only a distinguished *proposer* must be elected, to guarantee liveness [Lamport, 2001]. A reliable algorithm for election must use either randomness or real time [Fischer et al., 1985].

The mechanism used to ensure that no two proposals are issued with the same number is that different *proposers* choose their numbers from disjoint sets of numbers, and a *proposer* begins phase 1 every time with a higher proposal number than any it has already used [Lamport, 2001], since it remembers the highest-numbered proposal it tried to issue, which is stored safely.

Basic Paxos operates over several rounds, where each round has the following two phases [Lamport, 2001]:

1. *Phase 1*

   Prepare A *proposer* selects a proposal number n and sends a *prepare* request with that same number to a majority of *acceptors*.

   Promise If an *acceptor* receives a *prepare* request with number n greater than that of any *prepare* request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal (if any) that it has accepted.

2. *Phase 2*

   Accept If the *proposer* receives a response to its *prepare* requests (numbered n) from a majority of *acceptors*, then it sends an *accept* request to each of those *acceptors* for a proposal numbered n with a value v, which is the value of the highest-numbered proposal among the responses or any value if the responses reported no proposals.

   Accepted If an *acceptor* receives an *accept* request for a proposal numbered n, it accepts the proposal unless it has already responded to a *prepare* request number higher than n.

Since the progress of the algorithm can be compromised if, for instance, two *proposers* keep issuing a sequence of proposals with increasing numbers, a distinguished *proposer* must be selected as the only one to try issuing proposals and, eventually, it will choose a high enough proposal number to be accepted. However, this situation can arise when a leader or distinguished *proposer* fails and then recovers after the election of a new leader. The Ω oracle is used to perform the leader election and to ensure the liveness of the algorithm.

Typical deployments of Paxos require a continuous stream of agreed values acting as commands to a distributed state machine. There is a large overhead if a single instance of Basic Paxos is used to determine a single command.

Multi-Paxos is the most common deployment of the Paxos family. The optimization that leads to this variant is that *Phase 1* of the basic protocol may be ignored, and comes from the assumption that the *leader* is relatively

stable, which renders its execution unnecessary when the *leader* remains the same. This is achieved by including the instance number with each value. In this variant, the failure-free interval from the proposal to learning, is 2 message delays compared to the 4 on Basic Paxos.

Cheap Paxos extends Basic Paxos to tolerate f failures by using $f+1$ main processors and f auxiliary processors and by reconfiguring the system after each failure. However, liveness is sacrificed against the reduction of the number of processors, because the rapid failure of several main processors implies the stoppage of the system until the auxiliary processors can reconfigure the system. This is the sole purpose of the auxiliary processors, as they do not intervene in the protocol.

Fast Paxos extends Basic Paxos to reduce end-to-end message delivery latency. Whereas in Basic Paxos, the interval between the request of a client and the learning of the result is 3 message delays, Fast Paxos allows for only 2 message delays. This improvement is achieved if the *leader* has no value to propose and the client sends an *Accept!* message to the *acceptors* directly. Then, they would respond as in Basic Paxos, with *Accepted* messages sent to the *leader* and every *learner*. If the *leader* detects a collision, it sends *Accept!* messages for a new round and the algorithm proceeds as usual. In this case, the referred latency situation will comprise 4 message delays. There is an optimization that allows the *acceptors* to perform the collision recovery by themselves, if a recovery technique has been specified by the *leader*.

Generalized Paxos explores the relationship between the operations of a distributed state machine and the consensus protocol used to maintain the consistency of that system. When conflicting proposals could be applied to the state machine in any order, *i.e.* the operations contained in the conflicting proposals are commutative operations of the state machine, both can be accepted, avoiding the need for resolving conflicts and re-proposing the rejected operation.

This concept is used to generalize sets of commutative operations, that are tracked by the protocol to ensure that all the proposed commutative operations of one set are stabilized before allowing the stabilization of a non-commuting operation [Wiki Paxos].

The performance of Generalized Paxos is compared with that of Fast Paxos [Wiki Paxos]. So, to achieve the agreement on seven values, Generalized Paxos takes 10 messages delays whereas Fast Paxos takes from 15 to 17.

Byzantine Paxos extends Paxos to support Byzantine Fault Tolerance (BFT), which refers to, for instance, faults caused by participants lying, fabricating messages or colluding with other participants. To tolerate such type of failures, this protocol uses an additional message, called *Verify*, which is exchanged among *acceptors* to distribute their knowledge and verify the actions of one another. This step introduces an extra message delay, which is removed by the Fast Byzantine Paxos protocol, where the client sends commands directly to the *acceptors* [Wiki Paxos].

Paxos is not only a theoretical exercise but has been implemented in several production systems [Wiki Paxos]:

Chubby Lock Service keeps consistency of replicas in the event of a failure. Google's BigTable is built with Chubby, among other systems, and is used in such systems as Google Maps, YouTube, Blogger.com, etc.

Autopilot is the automatic data center management infrastructure developed within Microsoft over the last few years, which is responsible for automating software provisioning and deployment, system monitoring and carrying out repair actions to deal with faulty software and hardware.

IBM SAN Volume Controller (SVC) is a block storage virtualization appliance.

Raft

Raft [Ongaro and Ousterhout, 2013] is a consensus algorithm that adopts the replicated state machine approach. It decouples key elements of consensus, like leader election, log replication and safety, while enforcing a stronger degree of coherency to reduce the number of possible states. Raft is similar to other consensus algorithms, such as Viewstamped Replication [Liskov and Cowling, 2012; Oki and Liskov, 1988], but it stands out due to its strong leadership, as the leader concentrates as much functionality as possible, and its election is used as the first of two phases of consensus. Another key feature of Raft is its mechanism to support cluster membership changes, where the majorities of two different configurations overlap, allowing the cluster to operate normally during such transitions.

There are two different entities in the Raft protocol, servers and clients.

A server is always in one of three following states:

Leader Serves requests from clients and controls their application to the replicated log and state machine, while issuing periodic heartbeats to signal its liveness.

Candidate Corresponds to the transient state from *follower* to *leader*, during which the server starts an election trying to be elected as the new *leader*, by receiving a majority of votes.

Follower All servers start up in this state, which is passive, meaning that it awaits the contact of the *leader* or a *candidate* for a period of time equivalent to the election timeout, simply responding to those requests. If a server has not received any valid invocations from a *leader* or a *candidate*, during that period of time, it becomes a *candidate*. It will then increase its term value and reset the election timeout, assigning it a randomly selected value to help prevent split votes, and it will finally issue *RequestVote* RPCs in parallel to all known servers, starting a new leader election.

A Raft cluster is composed by several servers, being five a typical setup, allowing the system to tolerate two failures. In normal operation, there is a single *leader* on the cluster, being the remaining servers *followers*.

In the normal interaction of a client with a Raft cluster, it randomly selects a server and sends it a request. If that server is the *leader*, the request will be processed and the corresponding response returned to the client. Otherwise, the server sends the address of the most recent *leader* that it knows back to the client, which can then use it to contact the *leader* directly. In the event that the *leader* crashes, the client's request will time out and it will randomly select another server to interact with.

Raft uses the notion of term as an arbitrary period of time that starts with an election, where one or more *candidates* try to become the *leader*, but where at most one can take that role. If a *candidate* succeeds and becomes the new *leader*, it will keep that role until a new term is started. If there is no winner, a new election will be started, consequently on a new term. Terms are numbered with consecutive positive integers and are used as a logical clock [Lamport, 1978], allowing servers to detect obsolete information. Each server stores its current term number, which increases monotonically over time. This number is exchanged by communicating servers, allowing servers to update to the most recent value. If a server receives a request with an older term number, it is rejected, and its own current term number is sent back to the contacting server. If a *candidate* or a *leader* receives such a response, it immediately reverts to the *follower* state.

Raft uses the leader election as the first of two phases of consensus, using a heartbeat mechanism to trigger it.

1. When the election timeout of a *follower* elapses, because it has not received any valid invocation from a *leader* or a *candidate*, the server, assuming there is no valid *leader* on the cluster, becomes a *candidate*.

2. It will then increase its term value, reset the election timeout, by assigning a randomly selected value to help prevent split votes, and issue *RequestVote* RPCs in parallel to all known servers, starting a new leader election.

3. This candidate will be elected as the new *leader* after receiving a majority of votes from the servers comprising the cluster.

A *leader* uses this very same election timeout to trigger periodic heartbeats, that correspond to issuing *AppendEntries* RPCs that contain no log entries to all of its followers, in order to keep its authority. If a *leader* fails or becomes disconnected a new one is elected. All servers start up as *followers*, and wait to be contacted by the *leader* or a *candidate* for a period of time equivalent to the election timeout, which can take some initially configured value, or randomly selected when the server becomes a *candidate*, in order to prevent split votes.

When elected, the *leader* assumes full responsibility for managing the replicated log.

1. The *leader* accepts client requests, that contain some command to be executed by the replicated state machine, which is converted into an entry and added to its log.

2. Afterwards, it issues *AppendEntries* RPCs in parallel to its known *followers*, in order to replicate the entry.

3. When the new entry has been safely replicated, *i.e.* received a number of responses that is equal to the majority of the elements of the cluster, the *leader* applies the entry to its state machine and returns the result of that execution to the client.

4. The *leader* will then inform the replicas to commit that entry to their state machines in subsequent *AppendEntries* invocations.

The Raft algorithm ensures the replicated state machine safety property, which states that if any server has applied a particular log entry to its state

machine, no other server may apply a different command for the same log index.

The failure of a *follower* or a *candidate* is easily dealt by the Raft protocol, as any *RequestVote* or *AppendEntries* RPCs sent to it will fail. But when the server restarts as a *follower*, the RPCs will be delivered and processed correctly.

The authors of the Raft consider it an easier protocol to understand than Paxos [Lamport, 1998], fact supported by a user study where the majority of the enquired subjects found Raft easier than Paxos to implement and to explain [Ongaro and Ousterhout, 2013]. Another drawback of Paxos is the lack of a reference algorithm for multi-Paxos, as most descriptions fall on single-decree Paxos, or leave too many details to the implementer [Ongaro and Ousterhout, 2013]. Compared to Apache ZooKeeper, which is also leader-based, Raft is also a simpler protocol as it requires the implementation of fewer distinct operations and it also minimizes the functionality in non-leaders. For instance, in Raft, the log entries flow in a single direction, from leader to its replicas, whereas in ZooKeeper, entries flow both to and from the leader.

2.6 Discussion

Service-Oriented Computing has proven to be a very adaptable programming paradigm, even to an environment with scarce resources such as Wireless Sensor Networks [Mohamed and Al-Jaroodi, 2011], where Service-Oriented Middleware should fulfill some stringent requirements. The Devices Profile for Web Services (DPWS), as a standard specially targeted to enable the usage of Web Services by resource constrained devices, already provides several of the required features, such as dynamic, adaptive and auto-configurable architectures. Therefore, DPWS is a key standard for the implementation of the Internet-of-Things paradigm, and it has, for instance, become the enabler of the Smart Grid [Karnouskos, 2012], by allowing the interaction of entities with largely heterogeneous processing power, namely, smart meters and utilities back-end energy management systems [Karnouskos and Izmaylova, 2009], albeit indirectly through an hierarchical architecture. However, for a large amount of devices, in the region of some thousands, an hierarchically structured communication does not cope well with the generated traffic [Karnouskos et al., 2011]. This limitation, alongside with the targeted fault tolerance issues, proves that the traditional two-phase commit (2PC) mechanism does not suffice, as a failure in a system could stop the ongoing activity with numerous devices involved.

As transactional processing can be deemed heavy for the majority of the scenarios with resource constrained devices, the existing transactional standards for Web Services, such as WS-AtomicTransaction [WS-AT] and WS-BusinessActivity [WS-BA], both based on WS-Coordination [WS-C], can not be regarded as a solution to the problem. Other transactional protocols and standards, from which experience can be withdrawn and lessons learned, were also analyzed, such as RosettaNet [Alonso et al., 2004; Badakhchani, 2004], ebXML [Alonso et al., 2004], Business Transaction Protocol (BTP) [McGovern et al., 2006] and WS-Composite Application Framework (WS-CAF) [Little and Webber, 2003; Monsieur et al., 2007]. RosettaNet and ebXML can be considered vertical protocols as they focus on a business area and specify not only the transactional interaction, but also other aspects, like communication, for instance, where they predate some transversal standards for Web Services, such as SOAP, UDDI and WSDL [Alonso et al., 2004]. The other enumerated standards, BTP and WS-CAF, can be considered relatives of WS-Coordination, but were not as widely adopted.

Apparently, the way into the future could be an evolution of the WS-Coordination, WS-AtomicTransaction and WS-BusinessActivity standards, as they specify horizontal protocols built on top of SOAP, UDDI and WSDL, which could be enhanced by incorporating some of the lessons learned both with ebXML and RosettaNet. All in all, adversary trends of standards should harmonize to produce a global and coherent set of standards that would then be both simpler and easier to adopt [Alonso et al., 2004].

The interest associated with service coordination, provided by WS-Coordination, lies on the possibility of developing complex and composite services that ensure fault tolerance guarantees, using basic and simple services, that can not provide such assurances *per se*.

Transactional standards aside, the remaining standards for Web Services that are specially focused on fault tolerance include WS-ReliableMessaging [WS-RM], and WS-Reliability [WS-R], which can be considered the forefather of WS-ReliableMessaging, as many of its features were withdrawn from the experience with WS-Reliability. However these standards provide reliable communication only between two points, hence not enabling a multicast-typed communication.

Regarding this message exchange pattern, an extension for the usage of UDP Multicast with WS-Eventing [Gregorczyk, 2011] could help reduce the amount of traffic in scenarios where a single publisher must inform various subscribers on the occurrence of periodic events. However, the assurance of reliable delivery of events using a positive acknowledgment system would

cause an acknowledgment explosion in the publisher. And even in the case of well-known periodic events, where the usage of notification retransmission requests would be suitable, it could lead to a similar scenario if various subscribers do not receive the same event, since each will trigger a retransmission request which will ultimately accumulate on the publisher's side.

Besides the previously cited standards, there are several protocols that provide fault tolerance mechanisms for Web Services. However, there is a complete lack of standardization in this area, except for reliable communication and transactions as described before, which are not the only fault tolerance techniques that should be used to build a dependable architecture.

WS-Membership [Vogels and Re, 2003] proposed a framework that provides cooperating Web Services and activity monitors with a unified approach for tracking registered Web Services and for supplying membership updates to monitors using gossip-style communication, hence, promoting an highly robust and asynchronous membership information propagation mechanism with good reliability and scalability capabilities. However, it was not standardized and seems to have ceased to exist as little information can be found on the internet. Albeit the disadvantages of using epidemic failure detection, such as inefficiency when the size of messages grows proportionally with the number of participants, and bad behavior with massive concurrent participant failures, the detection of failed Member Services is very accurate. This framework is a good starting point for a future and improved membership service, which could be developed as the coordination of several basic services.

Service replication has already been attempted and used [Osrael et al., 2007b; Salas et al., 2006], but it is not suitable to ensure all the desired fault tolerance guarantees simultaneously with hard or even soft real-time requirements. In the event of a failure, the desired behavior for the system is to resume its normal operation as quickly as possible. Thus, failure recovery is another important issue that should be addressed in this project. An infrastructure that implements such capability for Web Services was presented in [He, 2004], but it does not address other types of failures apart from the basic *fail-stop*. Further research on this subject should focus on the effectiveness of this framework when dealing with other types of failures. It is clear that more emphasis should be put on standardization efforts for replication protocols, group membership services, group communication protocols, just to cite a few, in order to achieve more dependability of service-oriented systems [Osrael et al., 2007a]. The level of genericness and usefulness of a protocol are deemed essential for its standardization. For instance, the development of horizontal protocols like WS-Coordination, should be preferred

due to their generic nature.

Being consensus the basic problem involved in fault tolerance, since it depicts the issue of defining whether a process or node is up or down, and since gossip protocols are used to achieve reliable multicast, they can be considered as the building blocks for a large variety of distributed systems. So, they should be perceived in depth to fully apprehend their potential usefulness for the projected research. A generic consensus service [Guerraoui and Schiper, 2001] is extremely useful in achieving a simple methodology to build distributed systems with some complexity using simple and generic blocks that provide some basic service. Its applicability and usage in a Web Services environment would vary according to the intended use for a consensus mechanism. As Paxos [Lamport, 2001] is a family of diverse consensus protocols, it is interesting to verify the existence of any common blocks that allow the implementation of a generic Paxos service, that would provide, through some configuration, the most adequate Paxos variant for a certain scenario. Raft [Ongaro and Ousterhout, 2013] is similar to Viewstamped Replication [Liskov and Cowling, 2012; Oki and Liskov, 1988], but it stands out due to its strong leadership, as the leader concentrates as much functionality as possible, and it also supports cluster membership changes, allowing the cluster to operate normally during such transitions. The simplicity of the Raft protocol, when compared to Paxos, can increase its adoption in real systems, albeit its performance limitations due to the fact that all the operations must be checked and performed by the leader before being propagated to the replicas.

Gossip protocols are a very cost-effective means of achieving reliable multicast and, hence, very adequate to implement some services that need communication throughout a network, specially if it comprises a large number of nodes. At the moment, gossip-based algorithms are sufficiently mature to be used in the implementation of distributed systems although several challenges in this area remain unsurpassed [Kermarrec and van Steen, 2007a]. As stated in [Eugster et al., 2004], research on epidemic algorithms should be broadened beyond information dissemination to other areas such as content search, content-based publish/subscribe and also file sharing [Eugster et al., 2004]. With this intent, and being aware of the good performance of this type of algorithms, they would be an useful instrument for our future research focused on Web Services. More concretely, gossip protocols could be useful for implementing consensus, whether to reach a decision or to disseminate it also through a network, and various fault tolerance techniques, such as membership management that could resort to gossip in order to propa-

gate membership information and also to provide liveness schemes. There are other applications for gossip-based dissemination, like data aggregation and systems management [Kermarrec and van Steen, 2007a]. Furthermore, the conjugation of gossip protocols with service coordination could lead to communication models that would be innovative in the area of Web Services.

Chapter 3

Services

3.1 Gossip dissemination services

In this section, we present a service-oriented architecture for information dissemination based on existing standards and distributed gossiping. Gossiping is a lightweight approach for information dissemination that has inherent scalability and atomicity guarantees, while being simple, resilient, and frugal on resources. Our Web Services Gossip (WS-Gossip) framework provides operations that can be combined to architect a variety of gossip-style interactions, such as the *push* vs. *pull*, *eager* vs. *lazy*, and *infect-and-die* vs. *balls-and-bins* gossip variants, to address multiple applications and environments, and furthermore, can be integrated with different membership management strategies, through the included Peer Service that can be configured according to the scale and dynamics of each system. WS-Gossip also enables the usage of gossip by participants and their clients, while at the same time minimizing the impact on existing producers and consumers due to its adaptability.

3.1.1 Gossip service

Our proposal to address the scalability and reliability challenges of large DPWS deployments is to use a gossip-based dissemination protocol [Kermarrec and van Steen, 2007b]. Gossiping is inherently scalable, as it spreads the load across participants. Moreover, it is also inherently robust, tolerating message loss and participant crashes. This should have the increased advantage of allowing the usage of SOAP-over-UDP even if reliable delivery is desired, which is much less resource consuming than a full fledged HTTP binding over TCP. Moreover, by assuming the Web Services infrastructure, we take advantage of each gossiped unit of data being a SOAP envelope,

of the self-documenting nature of services through WSDL, and of further standards such as WS-Addressing and WS-Policy.

Providing comprehensive support for gossip-based information dissemination in Web Services, in a way that integrates with existing DPWS deployments, thus reduces to the following challenges:

- How to enable the usage of gossip by devices and clients, while at the same time minimizing the impact on producers and consumers of events, namely, regarding required middleware?
- How to support different peer discovery strategies, fit for different system scales and dynamics?

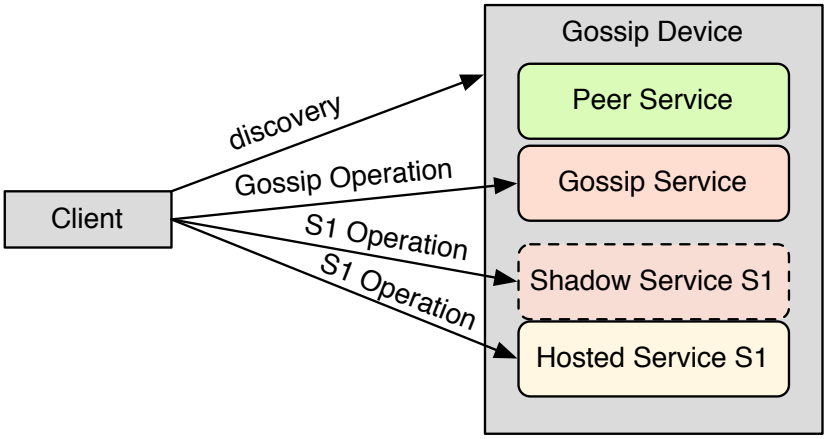


Figure 3.1: Overview of WS-Gossip architecture.

We address these challenges with a set of specifications of service port types, SOAP headers, and policy assertions that can be used to compose a variety of solutions. The general architecture of the proposed gossip dissemination framework is outlined in Figure 3.1 and works as follows. A manufacturer that intends to provide gossip dissemination in its devices can use a DPWS stack with gossiping support, by including the WS-Gossip framework, and annotate every service supporting gossip through WS-Policy assertions. As a consequence, a Shadow Service is created for each service where gossip is enabled. A single Gossip Service will also be hosted in such a device. Moreover, a Peer Service can be setup to provide an entry point to the set of target peers. Multiple Shadow and Gossip services can be attached to the same Peer Service, which might be hosted in a different device.

Both the original hosted service and its Shadow Service are advertised to clients that can use each of them independently. A gossip-aware client

can examine policy annotations in both these services and determine their relationship. A client may still address the original hosted service, thus maintaining compatibility with existing clients that are unaware of gossiping.

Assume for now a *one-way* or *notification* operation (*i.e.* input or output only) and *push* gossip [Karp et al., 2000]. Gossip dissemination can be performed using the Shadow Service or the regular Gossip Service.

In the case of the Shadow Service, gossiping is started when a client sends a SOAP message to a port of that service. Note that this service exports the same port type as the original hosted service, which means that a legacy client can still be used, simply by invoking the endpoint of the new service. Upon reception of this message, it is inspected to determine if it contains a WS-Gossip header. If not, default gossiping parameters are obtained, including gossip variant, fanout, peer scope (according to WS-Discovery), and target binding (HTTP or UDP). Gossiping is then initiated by adding the gossip information to the message header and relaying it to a number of peers and to the local hosted service, as outlined in Figure 3.2. When a gossip message is received, the gossiping interaction is continued by decrementing its hop count and by forwarding it to the selected peers. Note that such a message can be generated by a target device, as depicted in Figure 3.2, but it can also be generated directly by a gossiping-aware client. This allows a client to initiate gossiping in a custom scope or with custom parameters to achieve its own reliability and scalability trade-offs.

Such a gossiping-aware client can also use the regular Gossip Service for gossiping, by invoking an operation of that service directly, such as the *Push* operation, as depicted in Figure 3.3, which will include the message to disseminate as well as the desired parameter for the gossip dissemination. The targeted Gossip Service will then relay the included message to its known peers, through invocation of their *Push* operations, and pass the contained operation invocation to the adequate local service, if present in the same device.

The remainder of this section explains in detail the information contained in SOAP headers, how the Shadow and Gossip Services support multiple gossiping and SOAP operation styles, and how the target set of peers is discovered and managed.

Header information

As previously stated, the unit of information being gossiped is the SOAP envelope. Messages in a gossip interaction contain an entry in the SOAP header section of the SOAP envelope describing how to relay such messages. These are initialized by the initiator device, either within a Shadow Service

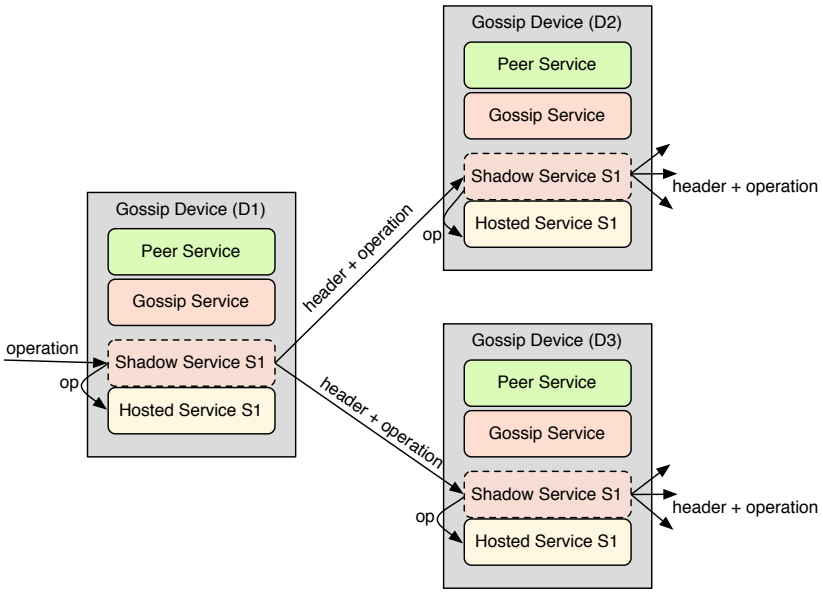


Figure 3.2: Gossip dissemination using the Shadow Service.

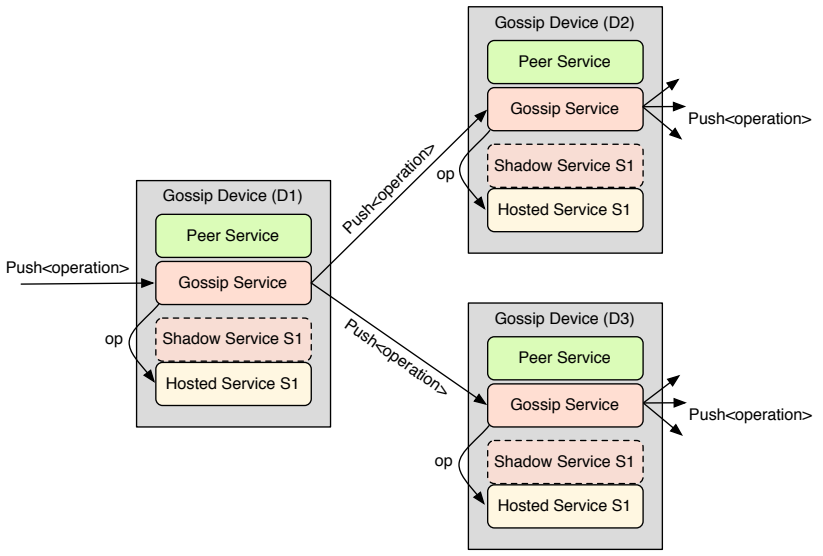


Figure 3.3: Gossip dissemination using the Gossip Service.

or by a gossip-aware client. Moreover, there is also the assumption of WS-Addressing providing a unique identifier for each message and support for asynchronous replies. Briefly, it contains the following information:

Scope/Type As defined by WS-Discovery, it implicitly describes the set of targets. Devices can be configured to relay messages only within a specific scope and type.

Fanout The number of peers to target in each interaction.

Hops The remaining number of hops. This must be decremented by each device that relays the message, and it is discarded when it reaches zero.

IdTTL The time that each device should buffer the message identifier for duplicate detection. If this is set to zero, the protocol degenerates to the *balls-and-bins* variant [Koldehofe, 2003].

DataTTL The time that each device should buffer the message itself for retransmission in lazy gossip variants. If this is set to zero, the protocol will never issue advertisements and will always use an eager variant.

Filter An optional item, specifying a rule to filter replies. Valid rules are configured by the deployer and advertised as policies by the Shadow Service.

Operation styles

SOAP and WSDL support several operation styles, like the typical client-server interaction (*i.e. request-response*), but it is also possible to have input-only operations (*i.e. one-way*), output-only operations (*i.e. notification*), and call-back operations (*i.e. solicit-response*). It is also possible that a *two-way* operation leads to multiple replies. These different operation styles allow WS-Gossip to support different gossip variants in addition to the previously described *eager push-style*, such as the *lazy* and the *pull* variants.

Gossiping in *one-way* and *notification* operations is handled as described previously: Upon reception of a message, it is propagated and no reply is expected. In *request-reply* and *solicit-response* operations, the message is propagated and then all replies received are propagated back to the initiator. This requires the initiator's address to be stored alongside with the message identifier used for duplicate detection during the specified **IdTTL**. Consider the following example: A *request-response* to query available disk space of servers in a data center. A client invokes the operation on the Shadow Service, which eventually reaches all targets. All responses then travel back along the

same tree implicitly created by the request message and will eventually reach the initiator.

An alternative is to make use of a filter, which can omit or aggregate replies according to a rule specified when gossiping is initiated. Consider the following example: The same *request-response* operation is used to determine which server has the most available disk space in a data center. This requires that upon deployment, devices are configured to support the maximum filter on the disk space query operation. A client invokes the operation on the Shadow Service, which eventually reaches all targets. Responses then travel back along the same tree implicitly created by the request message, but they are buffered and filtered such that only the maximum discovered downstream is returned by each peer. Each peer's reply is sent as soon as all its targets have replied, with a value or with a fault, or when a timeout expires.

Gossip styles

In addition to *eager push-style* gossip described so far, *lazy* and *pull* variants are supported as follows. Besides offering the same port type as the hosted service, the Shadow Service provides the same gossip port as the Gossip Service, which contains the following operations:

Push Alternative to directly using the interface. This allows a set of messages to be submitted in a single interaction.

PushIds Informs the target that a number of messages are locally available. These should then be requested using the *Fetch* operation.

Pull Returns currently buffered messages during a time interval specified as a parameter.

PullIds Variant of the previous operation, which requests identifiers instead of the actual messages. These can then be requested using the *Fetch* operation.

Fetch Returns currently buffered messages, as specified by a list of identifiers provided as a parameter.

Gossip variants can be achieved through the composition of the previous operations. Namely, *lazy push* is obtained by using *PushIds* instead of *Push* and then waiting for *Fetch* to be used later on selected identifiers. *Eager pull* is obtained by periodically invoking *Pull*. Finally, *lazy pull* is obtained by periodically invoking *PullIds* and then using *Fetch* on the resulting identifiers that are unknown.

The gossip variant chosen for each operation depends on configuration by the service deployer. In particular, the optimum configuration for *push* gossip is to use the *eager* variant for early rounds and then *lazy*. For *pull* gossip, the *lazy* variant is interesting for very large payloads. The combination of both *push* and *pull* is known to ensure rapid and robust dissemination of information [Birman et al., 1999; Karp et al., 2000].

3.1.2 Peer service

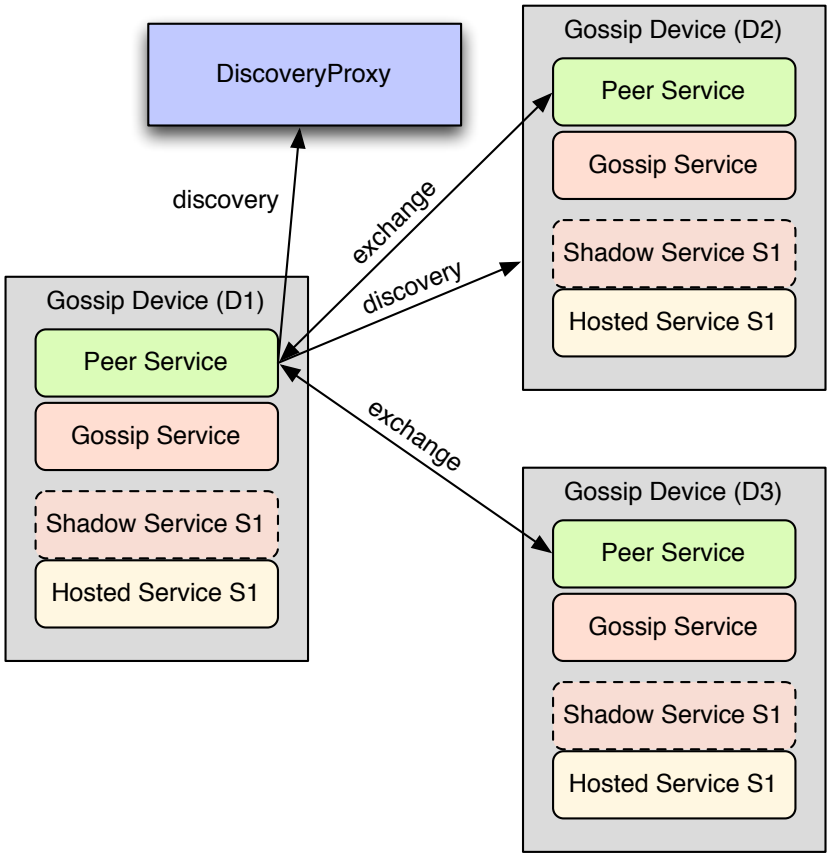


Figure 3.4: Overview of peer management.

By default, the usage of the WS-Gossip framework does not require an explicit peer management service. Instead, each gossip interaction is configured with a scope and a service type that can be used to discover the full set of reachable peers through WS-Discovery. This is most useful in scenarios where a discovery proxy device exists: In this case, when a set of peers is required for gossiping, it can be obtained efficiently by querying the proxy.

This leads to a configuration with centralized peer information while information dissemination is distributed, which is adequate for scenarios with low churn and relatively high messaging rate.

If such a proxy is not available, the usage of WS-Discovery in Ad-Hoc mode would lead to a large number of multicast messages that would most likely defeat the purpose of gossip. Instead, our proposal includes the Peer Service, which allows that information on discovered peers to be cached locally and exchanged with other Peer Service instances to implicitly create an overlay network using the Newscast protocol [Jelasity et al., 2003]. The structure of the stored peer information comprises a list where each service instance is represented by an entry that contains the following elements:

Address Corresponds to the service endpoint address.

Type Identifies the type of the service.

DeviceId Identifies the device where the service is hosted. This field is not applicable to services that are not associated with any device.

Heartbeat Counter that is incremented as messages, such as the invocation of the *Exchange* operation, are issued by the peer.

This information is exchanged among different devices and updated through the examination of WS-Discovery multicast messages issued by target services entering or leaving the network. Periodically, if a Peer Service instance has not received a request for exchanging its membership information during a certain time frame, it selects another instance to which it sends such a request containing the list of the known endpoints, through the invocation of the *Exchange* operation. Upon reception of such a message, the contacted instance returns to the requester its own list of known endpoints, and merges it with the received one.

The heartbeat counter of a service instance that never sends a new message, or eventually sends but without reaching a Peer Service instance, remains unchanged, implying that it will move towards the end of the membership list as the counter of other services is being updated and new services are discovered. That service instance will eventually be discarded when the cache of the Peer Service reaches the maximum configured size.

3.2 Consensus service

As we seek to provide the functionality of a coordination service, such as ZooKeeper, on the Devices Profile for Web Services (DPWS), this section

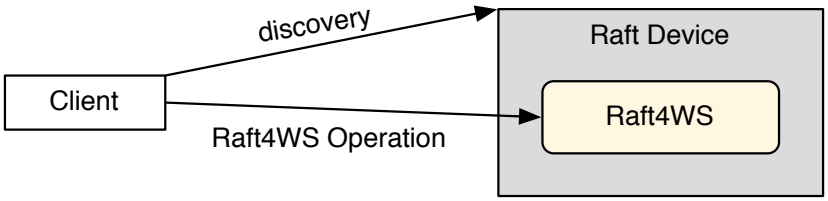


Figure 3.5: Overview of Raft4WS architecture.

describes the proposal of a Consensus Service, based on the Raft [Ongaro and Ousterhout, 2013] protocol, which enables the deployment of replicated Web Services that are able to tolerate both network and server failures. This proposal leverages existing DPWS components such as WS-Discovery for cluster membership maintenance and can easily be combined with WS-Eventing for event notification.

3.2.1 Raft service

Our proposal maps the Raft protocol to the DPWS environment, and therefore, we have implemented Raft for Web Services (Raft4WS), a Raft Service, on top of the Web Services for Devices (WS4D) Java Multi Edition DPWS Stack (JMEDS), which will be described throughout this section. According to the Raft protocol, our proposal considers two different entities: servers, which host an instance of the Raft Service, and clients, which invoke those instances. Figure 3.5 illustrates the possible interactions between a client and a server, hence a client must discover a server to contact, whether through configuration or dynamic discovery, and then it can invoke the operations available for clients.

Raft uses the leader election as the first of two phases of consensus, using a heartbeat mechanism to trigger it. Figure 3.6 illustrates all the steps involved in the leader election, which are described next. All servers start up as followers, and await the contact of a leader or a candidate for a period of time equivalent to the election timeout. When this election timeout elapses, because it has not received any valid invocations from a leader or a candidate, the server, assuming there is no leader on the cluster, becomes a candidate (Step 1). It will then increase its term value, reset the election timeout, by assigning a randomly selected value to help prevent split votes, and issue *RequestVote* RPCs in parallel to all known servers, starting a new leader election (Step 2). This candidate will be elected as the new leader after receiving a majority of votes from the servers comprising the cluster (Steps 3 and 4).

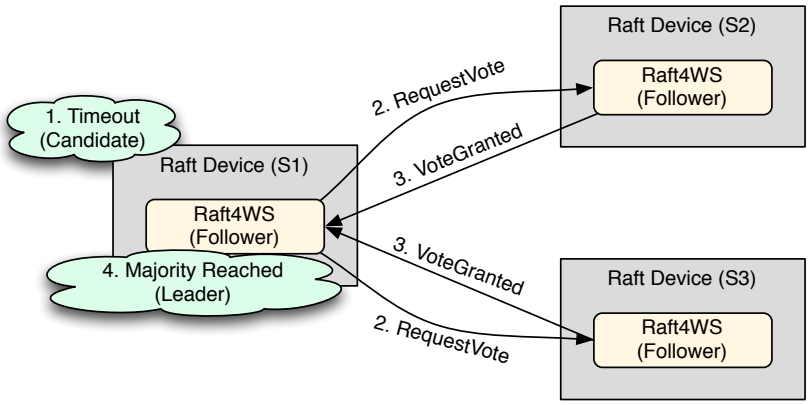


Figure 3.6: Overview of the leader election on Raft4WS.

A leader uses the very same timer, used to detect election timeout, to trigger periodic heartbeats, that correspond to issuing *AppendEntries* RPCs, without any log entries, to all of its followers, in order to keep its authority. If a leader fails or becomes disconnected, a new one is elected using the previously described leader election protocol.

When elected, the leader assumes full responsibility for managing the replicated log and attending client requests, as portrayed in Figure 3.7. Hence, it accepts a client request that contains a command to be executed by the replicated state machine which is converted into an entry added to its log (Step 1). Afterwards, the leader issues *AppendEntries* RPCs in parallel to its known followers, in order to replicate the entry (Step 2). When it has been safely replicated, *i.e.* received a number of responses sent by the followers (Step 3) that is equal to the majority of the elements of the cluster, it applies the entry to its state machine and returns the result of that execution to the client (Steps 4 and 5). The leader will inform the replicas to commit that entry to their state machines in subsequent *AppendEntries* invocations (Step 6).

Server

Using the DPWS terminology, a Server is a device with the type Raft_Device, and so, the terms Raft Device and Server will be used interchangeably throughout this section. The Server class contains five different entities, Log, Raft Service, ServerClient, TimeoutTask and the current state task. It also stores Raft specific parameters, such as the current term (*currentTerm*), the Server it has voted for (*votedFor*), the Server that is the current leader (*currentLeader*), the next index (*nextIndex*) and the match index (*matchIndex*)

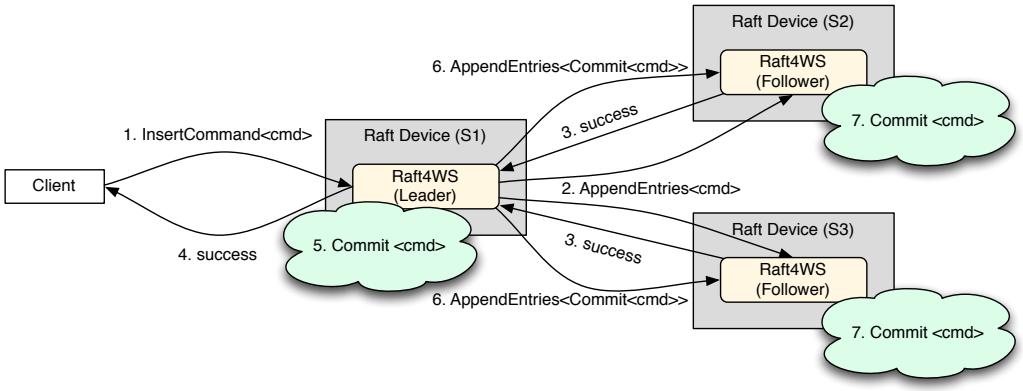


Figure 3.7: Overview of the insertion of a new command on Raft4WS.

for each replica or follower.

Essential for the replicated state machine approach, the Log keeps a list with all the entries, each represented by a LogEntry object, resultant of the commands inserted by clients, as well as the StateMachine and some variables, like *commitIndex* and *lastApplied*, which correspond to the highest log entry known to be committed or applied to the StateMachine, respectively. Each Server periodically runs a task to compare the values of these two variables. If *commitIndex* is bigger, *lastApplied* is incremented, and Log[*lastApplied*], *i.e.* the LogEntry with an index equal to *lastApplied*, is committed, by applying it to the StateMachine. The StateMachine provides operations for its initialization, termination, and for the insertion of a LogEntry returning a boolean value to convey the success of the insertion. Before describing the operations provided by Raft4WS, the structure of the LogEntry will be explained in more detail. Each new LogEntry is created by the leader and it stores the following parameters:

index Unique index assigned by the leader to be the size of Log (henceforth identified by *lastLogIndex*) incremented by one unit, which corresponds to the successor of the index of the last log entry, as the index of the first entry is 1.

term *currentTerm* of the leader.

uid Unique identifier extracted from the client's request.

command Operation to be executed on the StateMachine.

parameters Parameters for the operation defined by the command argument.

result Result of the execution of the LogEntry's *command*.

success Success of the execution of the LogEntry's *command*.

Besides these parameters, each LogEntry object also stores the number of follower responses needed to achieve the majority, according to the current size of the Raft cluster, and the respective lock, which just unlocks when the majority is reached. These responses are added to a LogEntry upon the reception of successful follower responses to the invocation of the *AppendEntries* operation containing LogEntry. After unlocking, the answer is sent back to the client that issued the request originally.

Raft4WS provides 3 different operations, which match very closely Raft's RPCs, and are available on every instance:

InsertCommand Invoked by clients to insert new commands in the cluster's replicated log. A request to this operation takes the following arguments:

uid Unique identifier generated by the invoking client to identify its request.

command Operation to be executed on the replicated state machine.

parameters Parameters for the operation defined in **command**.

The response of this operation comprises the arguments:

success Indicator of the success of the command execution by the leader, or always false if the Server is not the current leader.

result Result of applying the command if the Server is the current leader.

leaderAddress Conveys the contacted Server's *currentLeader* if it is a follower.

Upon receiving a request, the leader verifies if there is a LogEntry identified by **uid**, on which case its *result* will be sent right back to the client indicating a successful execution. Otherwise, the leader's Log creates the new LogEntry and sets the number of responses necessary to unlock its *result*, and the TimeoutTask is notified in order to trigger the invocation of the AppendEntries operation in order to propagate LogEntry to the followers. When LogEntry is unlocked, the response is sent back to the client, containing the **success** and the **result** arguments which convey the outcome of creating and committing LogEntry. If the Server is not the current leader of the Raft cluster, it responds right away to the client with **success** equal to false, and **leaderAddress** equal to *currentLeader*.

AppendEntries Invoked by the leader as an heartbeat, when there are no new entries, or to replicate new log entries on its followers. A request to this operation takes the following arguments:

term The *currentTerm* of the leader.

leaderId The address of the Raft Service of the leader.

prevLogIndex The index of the log entry that precedes the new ones.

prevLogTerm The term of the log entry identified by **prevLogIndex**.

entries The list of log entries to store, which will be empty in case the message is an heartbeat. Each entry is defined through its **index**, **term**, **uid**, **command** and **parameters** arguments.

leaderCommit Leader's *commitIndex*.

The response of this operation comprises the arguments:

term *currentTerm* of the targeted Server, for the leader to update itself.

success Boolean value indicating if the Log of the targeted Server contains the entry matching the values of **prevLogIndex** and **prevLogTerm**.

Independently of the success of this operation, the targeted Server returns its *currentTerm* on the reply. It starts by extracting the received **term**, and comparing it to *currentTerm*. If it is smaller, the Server will immediately send back a response with **success** as false. Otherwise, it checks if its Log contains a LogEntry with an index equal to **prevLogIndex**, and the *term* equal to **prevLogTerm**. If such a LogEntry does not exist, the **success** argument will be false and no further processing of the request takes place. However, if it does, the **success** argument of the response is set to true and the address of the leader, **leaderId**, will be copied to *currentLeader* if it is different from the previous value of *currentLeader*. The Server will notify its current state task by invoking its heartbeat method and will extract the entries from the request, reconstructing each LogEntry from its **index**, **term**, **uid**, **command** and **parameters** arguments, which will then be sent to the Log so they are appended to the existing entries. During this process, if there is any LogEntry in the Log that conflicts with a new one, by having the same *index* but different *terms*, it will be deleted as well as of all the subsequent existing ones. Any new LogEntry not in the Log is inserted in the corresponding *index*. Before returning its response to the leader, and if

the request processing has been successful so far, the Server compares the value of the **leaderCommit** argument to its *commitIndex*. If its *commitIndex* is inferior, it takes the minimum value between **leaderCommit** and its *lastLogIndex*. Finally, the response is sent back to the leader conveying the Server's *currentTerm* on **term**, and the **success** of the request.

RequestVote Invoked by candidates to gather votes from other Servers on the cluster. A request to this operation takes the following arguments:

term Candidate's *currentTerm*.

candidateId The address of the Raft Service of the candidate.

lastLogIndex The index of the last log entry of the candidate or *lastLogIndex*.

lastLogTerm The term of the last log entry of the candidate.

The response of this operation comprises the arguments:

term *currentTerm* of the targeted Server.

voteGranted Boolean value indicating if the Server has voted, or not, for this candidate to become the new leader.

The targeted Server compares **term** with its *currentTerm*. If it is smaller, the Server will immediately send back a response with *currentTerm* on the **term** argument, and the false value on the **voteGranted** argument. Otherwise, it compares **candidateId** with the Server's *votedFor*. If they are equal or if *votedFor* is null, the Server extracts and analyzes the remaining request arguments. It verifies if the candidate's Log is as up-to-date or more advanced than its own Log, by checking if the values of the **lastLogIndex** and **lastLogTerm** are greater than or equal to its corresponding parameters, *lastLogIndex* and the term of Log[*lastLogIndex*], respectively. If this conditions are confirmed, the Server sets its *votedFor* variable to **candidateId**, granting it the vote. After these verifications, the reply to the candidate conveys the Server's *currentTerm* value on **term**, and **voteGranted** will indicate if the Server has granted or not its vote to the candidate.

In the *AppendEntries* and *RequestVote* operations, the involved Servers always compare the value of the received **term** argument with its *currentTerm*. If the received value is higher, the Server sets its *currentTerm* to **term** and converts to the follower state, if it wasn't in that state previously.

After describing the Raft4WS Service and its operations, the remaining components of a Server are described, starting by its associated ServerClient

which can be used to detect other Servers, by listening to WS-Discovery multicast messages or issuing Probe messages to find other Raft Devices. All the detected Raft Devices will be queried to retrieve the *AppendEntries* and the *RequestVote* operation stubs, which are stored alongside with the matching Raft Service address, in case the current Server receives an invocation to its *InsertCommand* operation while it is not the leader and it is necessary to return the leader's Raft Service address to the client. This information is stored and indexed by the endpoint reference of the device. The detection of multicast Bye messages sent by a known Raft Device, makes the ServerClient remove the corresponding information on that device. Besides this function of maintaining the information on the cluster's elements, the ServerClient is the Server's component used to invoke operations on other Servers, such as *RequestVote*, when it becomes a candidate and starts a new election, or *AppendEntries*, when it is a leader and must signal its liveness, using heartbeats, or must replicate log entries on its followers. Such invocations are performed in parallel, by using a thread for invoking an operation on each known Server, following the guidelines of the Raft algorithm.

Each Server has a TimeoutTask thread that runs throughout its entire life, using a loop that is stopped when the Server shuts down. On each cycle, the TimeoutTask starts by checking if the state of the Server should be altered and performs the transition, if necessary. Afterwards, this task waits for a period of time corresponding to the configured election timeout value. When this time interval elapses, the TimeoutTask invokes the timeout method of the current state's task object. The timer will be interrupted, avoiding the mentioned invocation, by received invocations to the *AppendEntries* operation in both follower and candidate states, or to the *RequestVote* operation, in case the follower grants its vote, or when a majority of votes is received by the current Server while being a candidate.

According to the Raft protocol, a server can be in one of three different states, follower, candidate or leader, and it always starts its lifecycle as a follower. These states are represented by the FollowerTask, CandidateTask and LeaderTask classes which extend the ServerTask abstract class, making them share a basic interface with operations that are common to all the states, such as the reception of an invocation to the *AppendEntries* operation, the elapsing of the election timeout, or the termination of the current state. The signaling of the reception of an invocation to the *AppendEntries* is made through the heartbeat method. In the case of the FollowerTask the heartbeat method notifies the TimeoutTask to interrupt the currently waiting timer and skip to the next cycle, hence restarting the timer. The heartbeat methods of both the CandidateTask and the LeaderTask behave similarly, merely setting the Server's next state as follower, before notifying the TimeoutTask.

The invocation of the timeout method on the LeaderTask will cause the invocation of the *AppendEntries* operation on the known replicas. These invocations will convey any new entries received from the last invocation, or none if the leader's log has not been modified. The timeout method of the FollowerTask informs the TimeoutTask to make the Server transition to the candidate state. On the CandidateTask, the timeout method leads to the execution of a new election.

Client

The normal execution of a simple client for Raft4WS is to detect Raft Devices, whether by registering to listen to multicast WS-Discovery Hello messages from such devices, or to actively search for devices with such a type by issuing a multicast Probe message. If any Raft Devices are in the same network, they will respond to the client with a ProbeMatch message.

All the detected Raft Devices, whichever the used discovery mechanism, will be queried to retrieve the *InsertCommand* operation stub, which is stored with the matching device endpoint reference. The first detected Raft Device will be considered as the leader by the client, which will then become the target for its invocations of the *InsertCommand* operation.

Let us look into such an invocation in detail. A client prepares the invocation of the *InsertCommand* operation through the previously retrieved leader stub. It requests the creation of a Universally Unique Identifier (UUID) to the IDGenerator class provided by the WS4D JMEDS. This UUID, as well as the desired command and parameters are inserted in the request message that is then sent to the leader's *InsertCommand* operation. If the client's selected target is the current leader of the Raft cluster, the response will convey whether the creation and application of the corresponding LogEntry to the leader's StateMachine was successful and its result. Otherwise, *i.e.* if the targeted *InsertCommand* operation belongs to a Raft Device or Server that is currently a follower, the response will convey the false value as well as the address of the Raft4WS instance of the current leader. In this case, the client can then extract this address to reissue the invocation to the correct cluster leader, in order for it to become effective, as well as to send it further invocations.

Chapter 4

Results

This chapter presents the evaluation of the proposed framework, which comprises WS-Gossip and Raft4WS. Each of these services was evaluated using micro-benchmarks with settings as realistic as possible, according to each service's nature. WS-Gossip was tested in a simulator due to the large scale requirements of gossip protocols, which were impossible to replicate using the existing physical, or even virtual, machines. On the other hand, Raft4WS was tested using real physical machines, due to the reduced number of used replicas needed for the Raft protocol.

4.1 Gossip results

To evaluate the performance of the proposed approach, WS-Gossip was implemented, alongside with other evaluated protocols, using version 2 beta 3a of Java Multi Edition DPWS Stack (JMEDS), part of the Web Services for Devices (WS4D) project [WS4D]. The components of WS-Gossip, like the Gossip Service, the Shadow Service and the Peer Service, were implemented as regular hosted services, being able to coexist in the same device. For testing, we have also implemented a simple service that exports a simple *one-way* operation to set the value of a float variable, mimicking the propagation of temperature values. By minimizing the payload, we highlight the overhead of the protocol.

4.1.1 Experimental settings

Experimental evaluation is done using the Minha middleware test platform [Carvalho et al., 2011; Minha], which virtualizes multiple devices within a single JVM while simulating the performance characteristics of a real system.

It also allowed us to inject network faults to better assess the reliability of the various scenarios. Each test corresponds to the simulation of the run-time of a given number of devices collocated in the same LAN, in a single host with the following configuration: 64-bit Ubuntu Server 10.04.4 Linux, two 12-core AMD Opteron™ Processor 6172, 2.1GHz, 128 GB RAM, 64-bit Sun Microsystems Java SE 1.6.0_26.

The evaluation consists in executing a periodic event dissemination where a new value is propagated from a single producer device to a given number of consumer devices. A centralized managing device was used to control peer management and the execution of the test.

The following communication protocols/scenarios were analyzed:

WS-E A publish/subscribe communication protocol was selected as it is one of the most used event dissemination patterns. Hence, the WS-Eventing standard, as provided by the WS4D JMEDS framework, was evaluated using HTTP/TCP communication.

Multicast To assess multicast event dissemination, we resort to the SOAP-over-UDP protocol to use UDP Multicast.

WS-RM A simplified version of WS-ReliableMessaging, solely providing the *AtLeastOnce* message delivery assurance, was implemented and combined with SOAP-over-UDP, to avoid the overhead of TCP communication and to exploit its inherent reliable nature over UDP. In the execution of this scenario, each received message is acknowledged immediately after reception, and the producer waits for an acknowledgment during 50 milliseconds before resending the message. This value corresponds to the smallest time interval defined in SOAP-over-UDP for UDP retransmission.

WS-G In this scenario, the *push* gossip variant provided by WS-Gossip, using SOAP-over-UDP as transport, was evaluated.

AggWS-G This scenario extends WS-G with message aggregation capability. Hence, it enables each device to modify and resend a received message by processing the contained value and its own value, using the contained XSLT, which in this case contains the average function.

The execution procedure of each test comprised the following steps:

1. The manager and the producer devices are started.

2. The consumer devices are then started. In WS-E, they subscribe with the producer as soon as they are started. In WS-G and AggWS-G,

the manager informs each consumer of its neighbors, according to the configured fanout value, so they can convey new messages to them. In WS-RM, the manager informs the producer on the addresses of all the consumers so it is able to contact each one individually, in order to create a reliable message exchange sequence for each consumer. For all the scenarios, the manager verifies if all the devices started correctly before signaling the producer to start the dissemination.

3. The producer begins disseminating events periodically, which are propagated across the network.

4. The producer terminates and notifies the manager. In WS-RM, the producer closes the sequences that were used to communicate reliably with each consumer.

5. The manager informs, sequentially, all the devices about the file they should write their run statistics to.

The tests for each scenario consisted in 5 runs for each given number of devices, where 120 events were periodically emitted with an interval of 5 seconds. For all the scenarios, except WS-E, the tests were also run with communication losses, ranging from 0 to 20%, to compare the achieved reliability and latency degradation.

The interval between the initial emission of a message and its reception by a consumer was measured in nanoseconds since Minha enables the execution of all the intervening devices inside a single JVM on a single host. The sampling of the instant of emission was performed right before the producer sends a message, and the reception time measurement was done in the first operation of the method invoked to deal with a new message at a consumer.

In WS-G and AggWS-G, the used values for the fanout parameter were computed according to [Eugster et al., 2004], taking into account the number of devices, as well as an expected error rate (e) of 5% and a delivery assurance (p) of 99%, ranging from a value of 8 for 10 devices to 11 for 250 devices. In these very same scenarios, the publisher is randomly selected from all the nodes, contrarily to the other scenarios where the publisher is the first device.

4.1.2 Results and discussion

Results presented in Figures 4.1 to 4.7 are the average of all 5 runs for each setting. For latency measurements, the first and the last 10 iterations were discarded in order to minimize the effect of Java JIT compilation, although it also masks the delay of TCP connection establishment in WS-E.

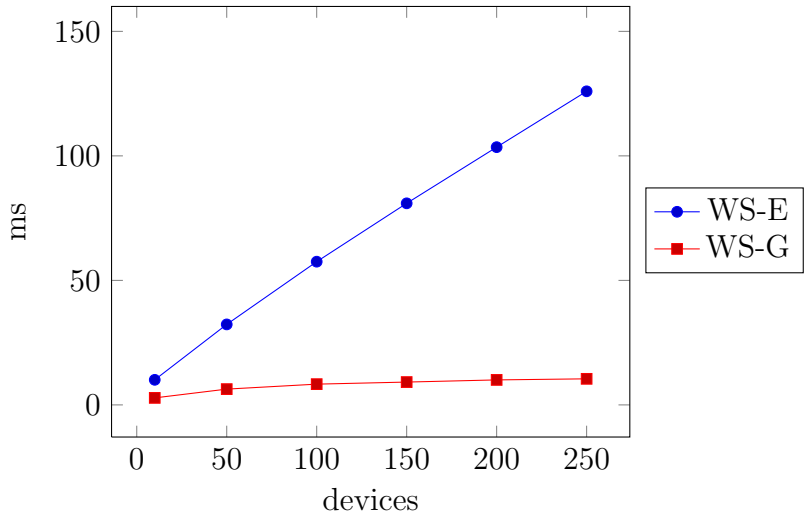


Figure 4.1: WS-E vs. WS-G (latency).

In Figure 4.1, the message delivery latency of the WS-E grows linearly with the number of targets, from 10 to 125 milliseconds, whereas that of WS-G is very small and grows very slowly, from just under 3 to 10 milliseconds. This is justified by scattering the load of propagating a message through an entire network, by the devices on that network, instead of overloading a single device, such as the publisher in other scenarios. Figure 4.2 presents the logarithmic growth of the average number of hops a message goes through from emission to reception in WS-G, confirming that the gossip protocol scales logarithmically with system size.

Figure 4.3 compares WS-RM and WS-G, with both none and 10% communication losses. It can be seen that when there are no losses, the two protocols start with a similar performance in terms of latency, but as the number of targets increases the latency of WS-RM starts detaching from WS-G and grows linearly from 6 to 130 milliseconds, whereas the latency of WS-G remains under 10 milliseconds at all times. With 10% communication losses, WS-G seems to be slightly affected as it suffers a negligible latency increase of around 0.1 milliseconds throughout the entire series, whereas in WS-RM latency also seems to increase linearly, but faster than without losses, between 55 and 218 milliseconds, probably due to the occurrence of losses and the increasingly overall impact of retransmission delays.

The effects of message losses on the latency of WS-RM are shown in Figure 4.4. WS-RM (1% loss) and WS-RM (5% loss) are very close to the baseline, suffering an increase in latency from around 0.3 to 3 milliseconds, for WS-RM (1% loss), and between 15 and 27 milliseconds, for WS-RM (5%

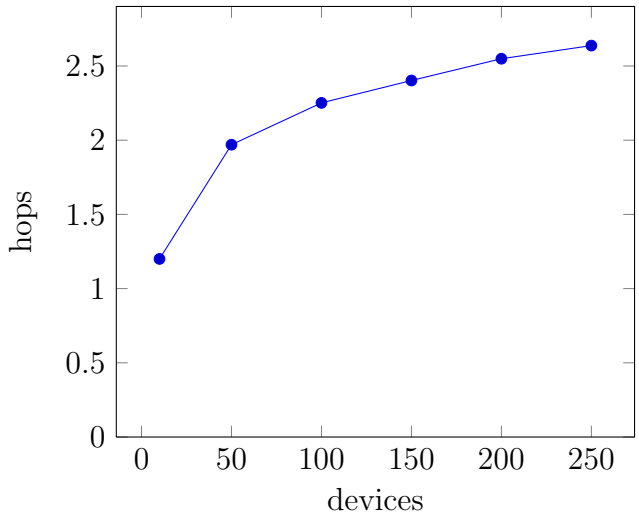


Figure 4.2: Average hops to delivery in WS-G.

loss). As mentioned previously for Figure 4.3, the latency of WS-RM (10% loss) is largely increased compared to the baseline. WS-RM (20% loss) should be stressed as an extreme case, where latency starts at 290 milliseconds for 10 devices, finishing at 450 milliseconds for 250 devices. This really shows the limitations of WS-RM when dealing with a large amount of communication failures.

The latency of the both gossiping scenarios can be seen in Figure 4.5, and portrays the overhead introduced by the usage of aggregation. Comparing both baseline scenarios, the increase in latency goes from 2 milliseconds for 10 devices, to 13 milliseconds for 250 devices, which can be considered as a small price to pay to obtain aggregation capabilities in such a lightweight protocol. For both gossiping scenarios the latency increases ever so slightly throughout the series with the increase of communication losses, ranging between 0.5 to 1.5 milliseconds, from WS-G (0% loss) to WS-G (20% loss), and between 2.5 to 3.5 milliseconds, from AggWS-G (0% loss) to AggWS-G (20% loss).

The latency on the various Multicast scenarios is depicted in Figure 4.6, showing very small values which do not seem to be largely affected by the increase on the number of devices on the network. However, the latency for Multicast (0% loss), which remains around 0.7 milliseconds, is higher than that of Multicast (1% loss), which remains around 0.6 milliseconds. The series for the other scenarios diverge a little bit in behavior, but all seem to indicate a trend for the latency to remain under the millisecond barrier, which could be verified with further experiments.

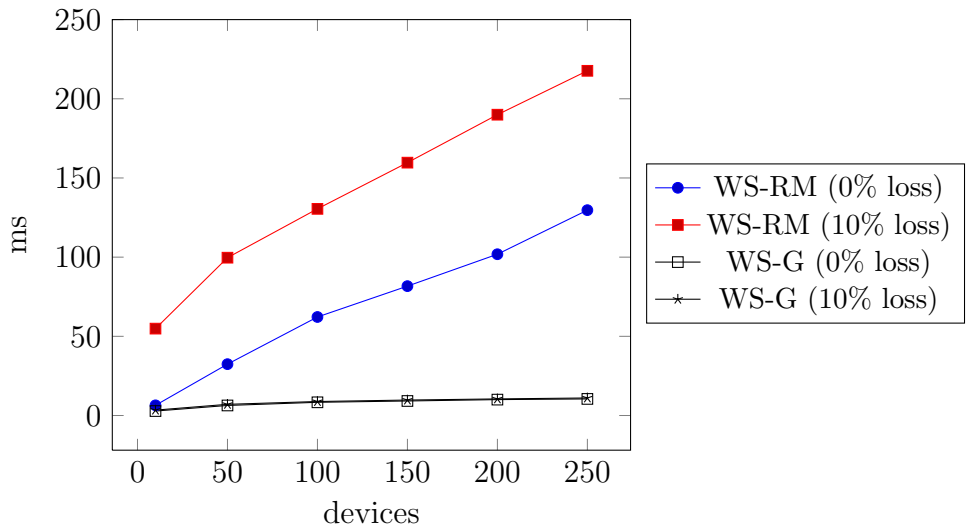


Figure 4.3: WS-RM vs. WS-G (latency).

Figure 4.7 presents the message delivery rate for the Multicast scenario, where, as predicted, the scenario without any communication losses achieves 100%, and the message delivery rate decreases as the communication error rate increases. These values are, in average, 97%, for Multicast (1% loss), 89% for Multicast (5% loss), 80% for Multicast (10% loss) and 63% for Multicast (20% loss). The message delivery rate of the other scenarios is not presented graphically since it is 100% both in WS-E and in WS-RM, and in WS-G and AggWS-G, it is greater than 99,9% in all runs, and most frequently 100%, even with 10% of communication losses, which corresponds to the double of the expected 5% value, which was used to compute the fanout parameter for these scenarios.

The presented results show the limitations of WS-Eventing when dealing with multiple devices. As a publisher is obliged to maintain, or restart, a TCP connection with each subscriber, this proves to be very resource consuming as the latency increases very clearly with the number of subscribers. Another evidence of these results, is the limitation of the applicability of Multicast to faulty scenarios, as it cannot cope with and recover from any communication error, albeit its high speed of delivery which can be essential in some cases. Our intention to use WS-ReliableMessaging on top of UDP was to tolerate communication failures, and expectations were that it would perform well. However, the results show a different behavior: for small error rates, between 1% and 5%, or even 10%, the overhead introduced by resending the lost messages in order to assure they are delivered to their destination, can be

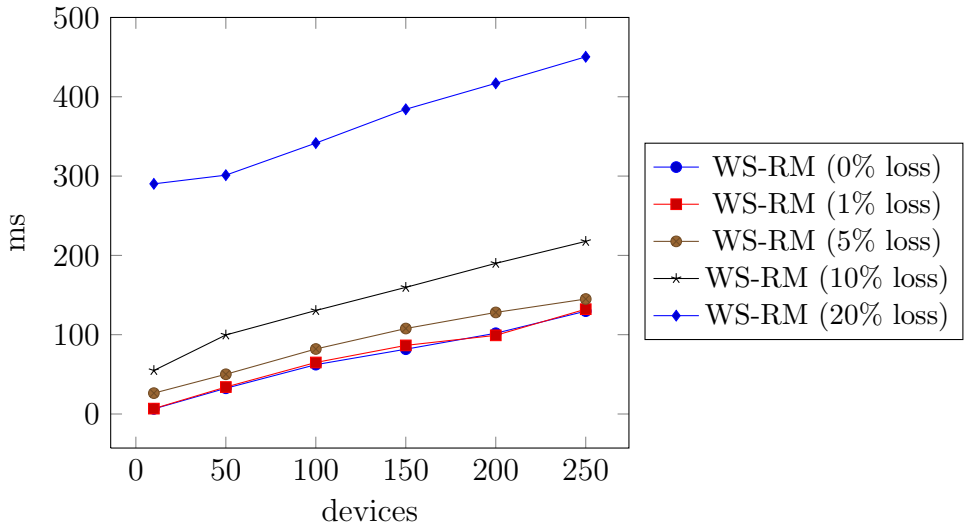


Figure 4.4: WS-RM (latency).

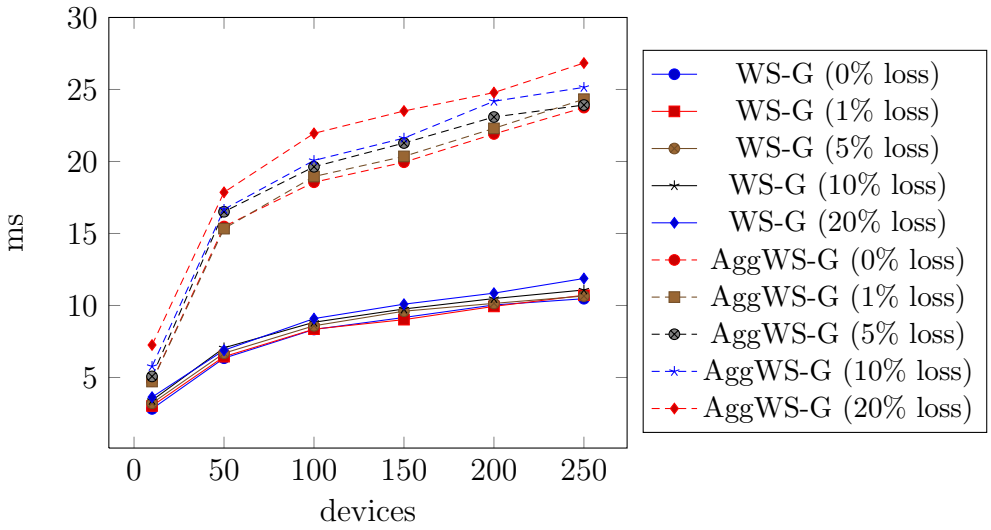


Figure 4.5: Push (WS-G) vs. Aggregation Push (AggWS-G) (latency).

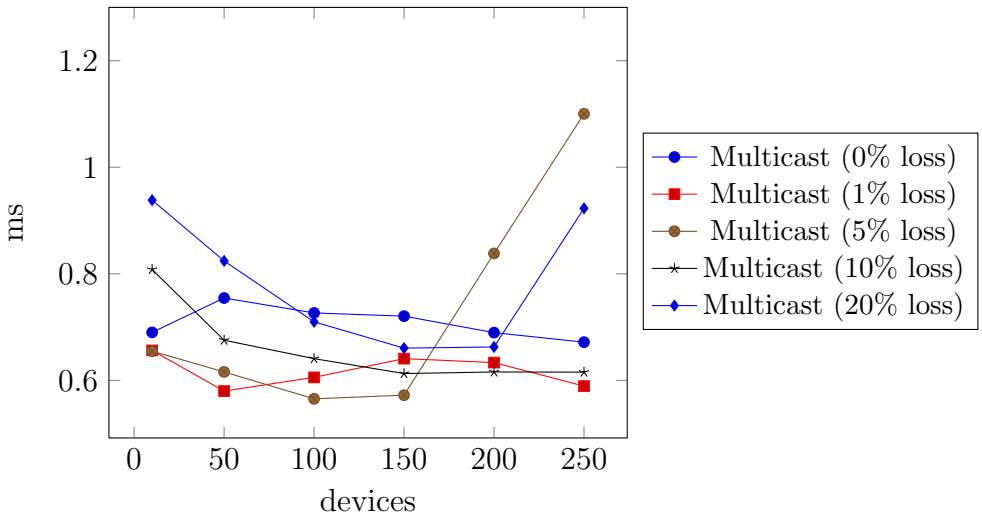


Figure 4.6: Multicast (latency).

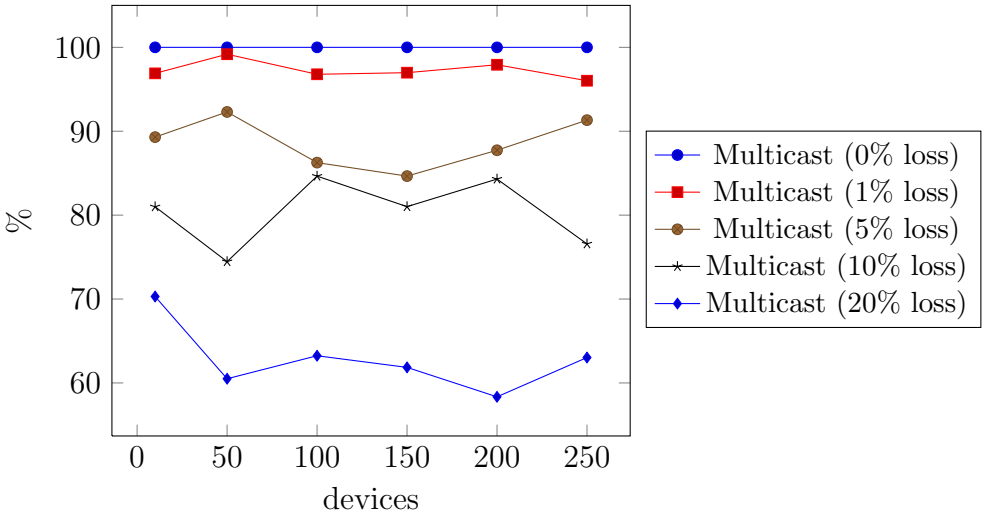


Figure 4.7: Multicast (message delivery rate).

considered acceptable. However, for a 20% communication error rate, this overhead is very large, varying between 284 and 320 milliseconds throughout the series. From the comparison of WS-G with other scenarios, it becomes clear that the performance of gossip protocols is not too much affected by the increase on the number of destinations, demonstrating how well they can scale. The results also show clear evidence of the intrinsic reliability capabilities of gossip protocols, as their performance remains pretty much the same, even in scenarios with a large amount of failures.

To conclude the analysis of the results, consider an environment with n devices. Scenarios such as WS-E or WS-RM, where a single producer is responsible for contacting all the devices, it will always have to send n messages for each event, whereas gossip peers will send a number of messages equal to its fanout, thus spreading the load throughout the network, which results in a lower load on the producer for cases where $n > f$.

4.2 Consensus results

To evaluate the performance of Raft4WS, which was implemented using version 2 beta 10 of the Web Services for Devices (WS4D) [WS4D] Java Multi Edition DPWS Stack (JMEDS), we compared it with ZooKeeper in similar scenarios with a single server, and three or five servers, to process the requests of 1, 5, 10 or 25 clients. Five runs were executed for each scenario and the average of their results is presented.

We have leveraged WS-Discovery's inclusion on DPWS by having a test manager listen to multicast announcements of clients and servers entering the network, to determine if all the intervening components were up and running in order to start the execution of test. Hence, the execution procedure of each Raft test comprised the following steps:

1. The manager and the clients are started on the same machine.

2. Each server is then started on its own machine.

3. The manager waits until detecting that all the expected clients and servers, for the current test, have been started. The manager will then select the leader server and contact each client to inform about it and also on the number of iterations to execute. The manager will also inform each server about its state, if it is a leader or a follower, its neighboring servers, and the value of the election timeout, which is randomly selected from 150 to 300 milliseconds. After conveying all the relevant parameters, the manager requests all servers to start

running and waits for twice the value of the election timeout, before subscribing to the event signaling the end of the workload on all the clients. Afterwards, it requests the start of the workload on all clients.

4. The clients start running the specified iterations where they invoke the *InsertCommand* operation on the leader server.

5. The clients terminate and notify the manager, which waits until all the clients have notified it.

6. The manager informs all the clients and servers on the name of the file they should write their run statistics to.

7. All clients and servers terminate after writing the statistics.

The effects of failing servers were also evaluated, until the maximum number of tolerated failures, as the service should become unavailable in order to guarantee its correctness. For this purpose, the manager was configured to cause a failure at 500 milliseconds after the start of the workload, which could be a follower or the leader in the scenarios with 3 servers. These two scenarios will be hence forth identified by 1 Follower (1F) and 1 Leader (1L), respectively. In the failure scenarios with 5 servers, a follower failure is introduced at the same instant of time, *i.e.* 500 milliseconds, followed by another failure, after another 500 milliseconds, which can be another follower or the leader. These will be henceforth identified by 2 Followers (2F) and 1 Follower 1 Leader (1F1L), respectively. The baseline scenarios, when compared with the ones with failures, will be identified by 0 Errors (0E) to better distinguish them.

In order to better compare the behavior of a client reconnecting to the cluster after losing its current connection, we have mimicked the behavior of the ClientCnxn class as provided in ZooKeeper. When a loss of connection is detected, it waits during a period of time, whose dimension is randomly generated until a maximum of 1 second, before attempting to connect to another server.

We compared Raft4WS with version 3.4.5 of Apache ZooKeeper. The configuration parameters used for all the settings were the following: a tickTime of 2000 milliseconds, which corresponds to ZooKeeper's basic time unit or heartbeat interval; an initLimit of 5, which means ZooKeeper servers have 5 ticks to connect to a leader; and a syncLimit of 2 ticks, which is the maximum delay of the state of a ZooKeeper server compared to the quorum's leader. The procedure for each ZooKeeper test was exactly the same as described for the Raft4WS tests, with the exception of the interactions between

the manager and the servers, since we did not want to modify the code of the ZooKeeper servers. Hence, each ZooKeeper server was initialized through the supplied zkServer bash script, and the test manager and the clients were initialized after waiting for the time corresponding to the initLimit parameter.

4.2.1 Experimental settings

The experimental evaluation of our implementation of the Raft protocol compared to the Apache ZooKeeper was performed on six identical hosts connected to the same LAN, with the following configuration: 64-bit Ubuntu 12.04.4 Linux, Intel^{R} Core^{TM} i3-2100, 3.10GHz, 8GB RAM, 64-bit Java^{TM} SE 1.6.0_27. One machine was exclusively used to run the manager and all the clients, whereas each of the remaining machines was used to run a single Raft4WS or Zookeeper server, in sets of 1, 3 or 5 servers according to the tests' settings.

Each client executed 120 iterations without any interval, where each iteration consists on invoking the insertion of a new command on the leader server in the case of Raft4WS, or a randomly selected server in the case of ZooKeeper, as it is the default behavior of the ZooKeeper client. Each command contains a unique identifier, defined by each invoking client, as well as the actual command and the corresponding parameters. The leader, after receiving such a request, creates the corresponding entry on its log and invokes the *AppendEntries* operation of Raft4WS on its known replicas, to propagate the new entry. The leader will only respond to the client when the majority of its replicas has replied successfully to this invocation. For the execution of the Raft4WS tests, the selected state machine was the one built using Berkeley DB.

In the case of ZooKeeper, the insertion of a command corresponds to the creation of a file with the unique identifier as its name, with the command and the parameters as its contents. The ZooKeeper server replies with the full file path to the client. Before each run, all the created files were deleted, in order to start with an empty file-system. For latency measurements, the first and the last 10 iterations were discarded in order to minimize the effect of Java JIT compilation, although it also masks the delay of TCP connection establishment, whereas throughput takes into account all the 120 iterations, where clients issued requests, and the time it took servers to process all of them and to reply back.

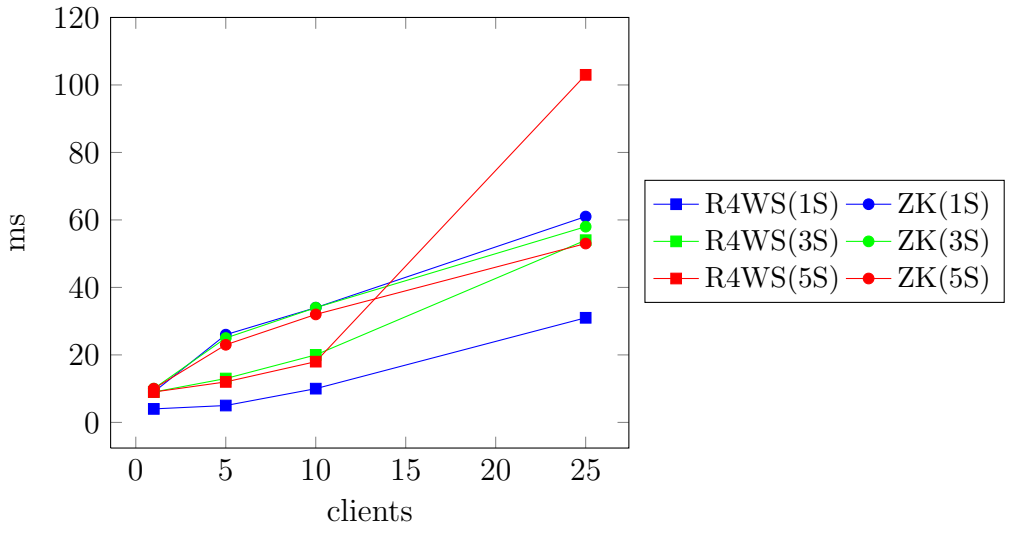


Figure 4.8: Raft4WS vs. ZooKeeper (Latency).

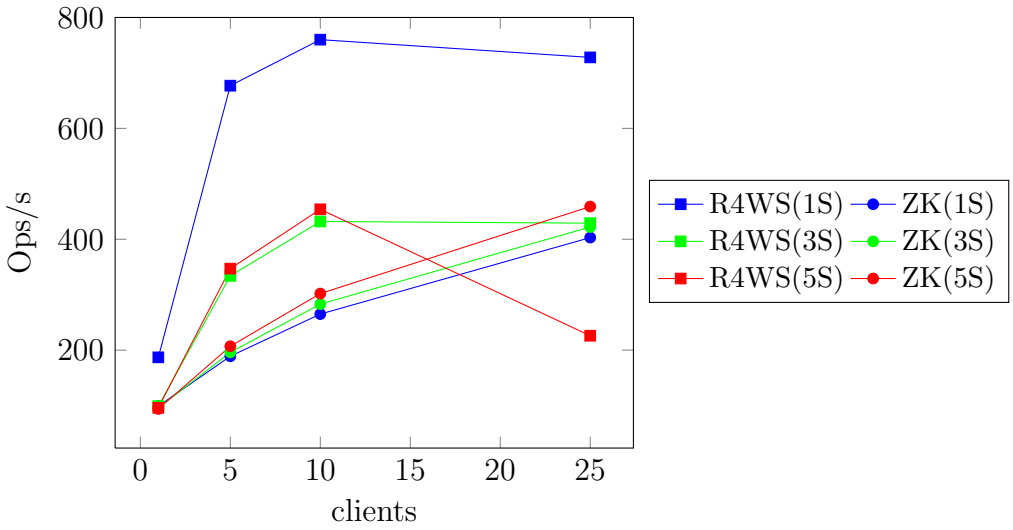


Figure 4.9: Raft4WS vs. ZooKeeper (Throughput).

4.2.2 Results and discussion

In Figure 4.8, the latency of all the scenarios grows linearly with the number of clients, with the exception of Raft4WS with 5 servers, where latency seems to increase exponentially from 10 to 25 clients, possibly, showing signs of saturation of the cluster's resources. The latency of the baseline scenario, *i.e.* Raft4WS with a single server which corresponds to the non-replicated service, is, as expected, noticeably inferior to the remaining scenarios, independently of the number of clients. It is important to notice that the latency of the various ZooKeeper scenarios, decreases as the number of servers increases. Another important fact is that the latency of the Raft4WS scenarios with both 3 or 5 servers is very similar and inferior to the corresponding ZooKeeper scenario, with the exception of the case mentioned before, *i.e.*, Raft4WS with 5 servers and 25 clients, which reaches an average of around 103 milliseconds. Figure 4.9 shows the throughput in operations per second that the servers can fulfill in the various scenarios, which is closely related with the latency values depicted in Figure 4.8. Regarding the ZooKeeper scenarios, one can see that the throughput increases with the number of servers, which supports the known parallelism and high-availability capabilities of ZooKeeper, as clients can be handled by any server of the cluster. On the other hand, as the Raft protocol only allows the leader to satisfy client requests, which consist in the insertion of commands in these tests, the scalability of this protocol suffers from this limitation, as the leader can easily become overloaded since it processes all the updates to the state machine, by propagating them to the followers, and needs to respond to the connected clients.

The effects of a failed server, at 500 milliseconds, in a cluster with 3 servers can be observed in Figure 4.10 and Figure 4.11. The average latency of the ZooKeeper scenarios is always higher than in the corresponding Raft4WS scenario, and, in all of them, it increases linearly with the number of clients. In terms of throughput, it seems to increase linearly in the ZooKeeper scenarios opposed to the Raft4WS ones, where a peak is reached in the runs with 10 clients, with the throughput decreasing slightly or stabilizing afterwards, which seems to show that the plateau of the processing capabilities of the leader has been reached.

The failure of a ZooKeeper server, always increases latency, reducing throughput consequently. Whereas the failure of a follower only deteriorates the performance slightly, from 2 to 5 milliseconds in latency and from 10 to 30 operations per second in throughput, the failure of the leader introduces a penalty of 20 to 30 milliseconds in latency and of 70 to 120 operations per second in throughput.

The effects of a failed Raft4WS server vary. Scenarios with a failed fol-

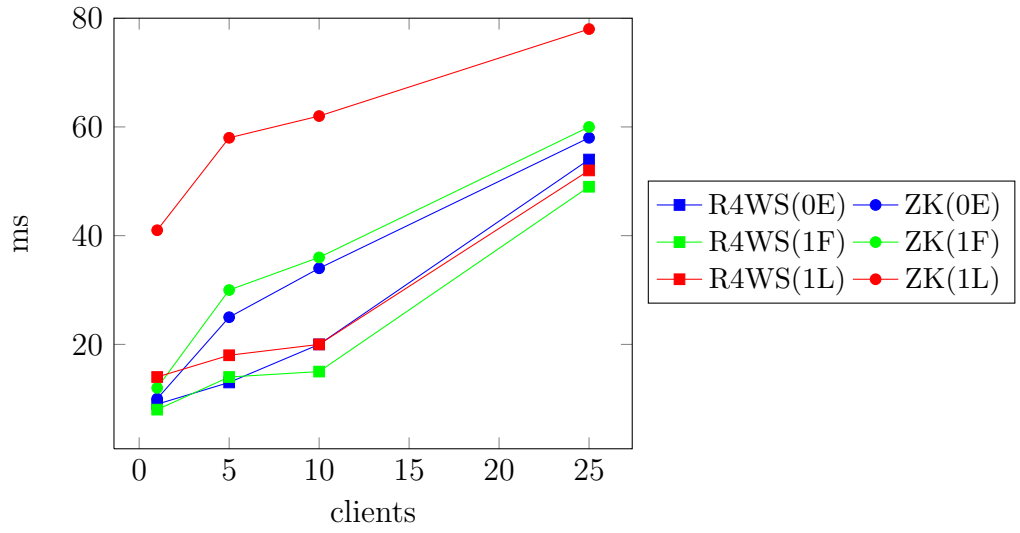


Figure 4.10: Raft4WS vs. ZooKeeper with 3 servers and a failure at 500 ms (Latency).

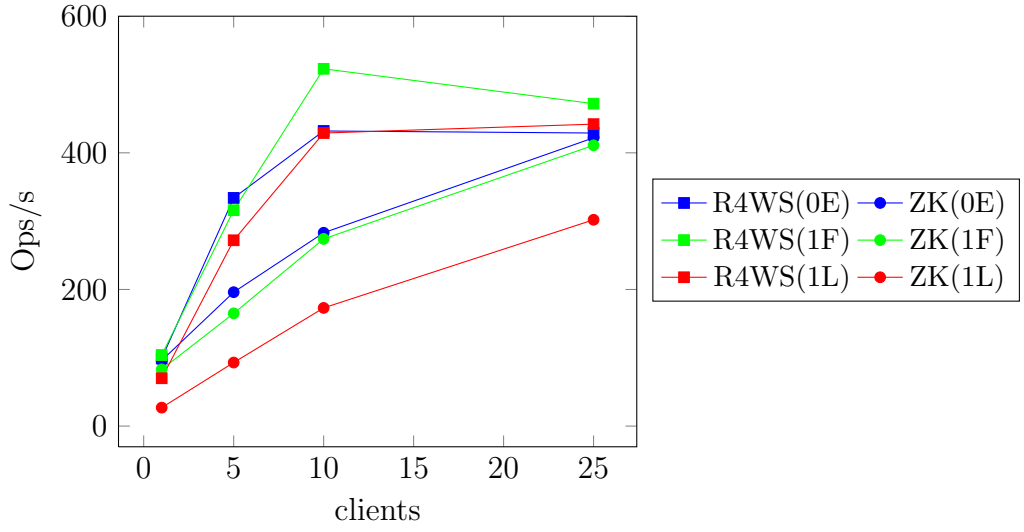


Figure 4.11: Raft4WS vs. ZooKeeper with 3 servers and a failure at 500 ms (Throughput).

lower have better performance than the baseline, with latency decreasing between 1 and 5 milliseconds, and increasing throughput between 5 and 90 operations per second, except for 5 clients, where it is slightly worse. This can be explained by the smaller number of messages that the leader needs to send, as it only needs to contact a single follower, instead of two as in the baseline. The scenario with the failed leader implies that all clients connect to the newly elected leader, to fulfill its requests, having worse performance, as occurs more distinctively for 1 and 5 clients, and in a smaller degree for 10 clients. For 25 clients, its performance is better than the baseline, which can be explained with the same phenomenon caused by a failed follower, *i.e.*, as the leader is killed, a follower will eventually be elected as the new leader, which will need to contact the single follower remaining in the cluster, hence issuing a smaller number of messages. This will certainly counterbalance the penalty introduced by the failure of the leader, which causes clients to connect to the new leader.

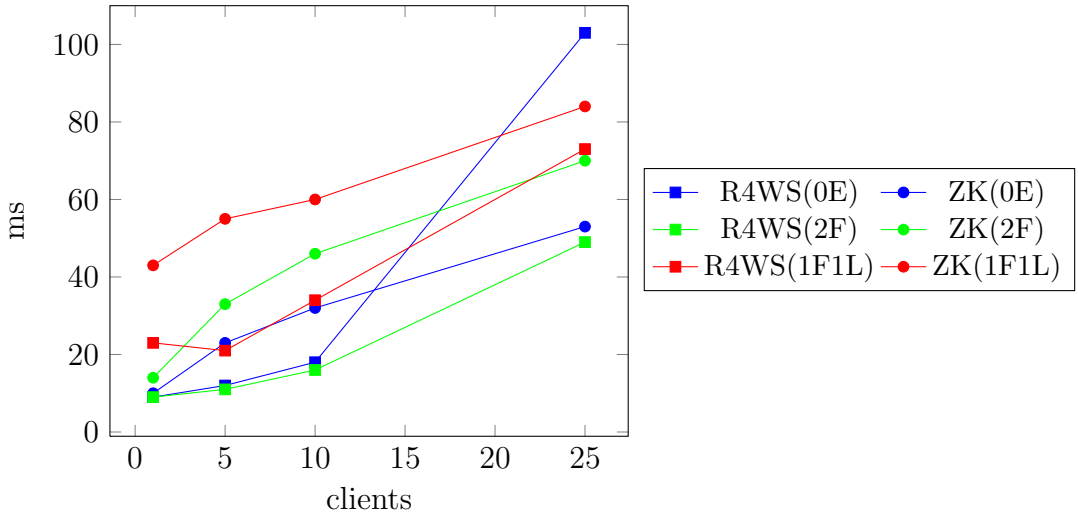


Figure 4.12: Raft4WS vs. ZooKeeper with 5 servers and two failures at 500 and 1000 ms (Latency).

Figure 4.12 and Figure 4.13 portray the influence of 2 failed servers, a follower at 500 milliseconds, and another follower or the leader at 1000 milliseconds, in a cluster with 5 servers, reaching the maximum number of tolerated failures. As in the cluster of 3 servers, the average latency of the ZooKeeper scenarios is always higher than the corresponding Raft4WS scenario, which increases linearly with the number of clients in all the scenarios. The only exception, as mentioned previously in the comments to Figure 4.8 and Figure 4.9, was the baseline Raft4WS scenario with 25 clients.

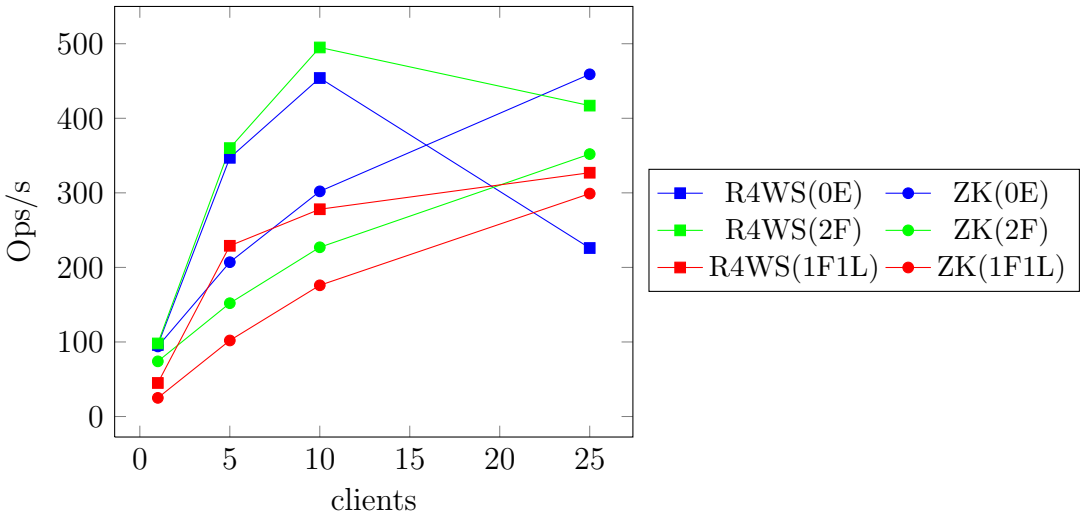


Figure 4.13: Raft4WS vs. ZooKeeper with 5 servers and two failures at 500 and 1000 ms (Throughput).

Failed ZooKeeper servers always introduce a performance penalty, due to the need of clients, that were connected to the failing server, to reconnect to a different one in order to invoke their requests, in both failure scenarios. Latency increases between 4 and 17 milliseconds, and throughput decreases between 9 and 31 operations per second, in the 2F scenario. The 1F1L scenario introduces an aggravated penalty, as latency is around 30 milliseconds higher and the throughput decreases between 20 and 130 operations per second compared to the baseline.

The effects of failed servers in a Raft4WS cluster with 5 servers vary in a similar way to what was observed for 3 servers. The failure of 2 followers improves the performance, by reducing ever so slightly the latency, and increasing the throughput between 2 to 40 operations per second, with the exception of 25 clients, where the cluster is clearly saturated in the baseline, which worsens its overall performance dramatically. This same setting for the baseline is the only one where the 1F1L scenario has a better performance than the baseline. On the rest of the settings, the 1F1L introduces an overhead varying from 9 to 16 milliseconds in terms of latency and a reduction of around 50 to 180 operations per second in throughput.

To sum it all up, the performance of Raft4WS is always better than that of ZooKeeper in similar settings, with the exception of Raft4WS with 5 servers, which shows signs of resources saturation. However, Figure 4.9 shows a trend where the throughput of ZooKeeper is still increasing, which could continue past the maximum tested 25 clients, whereas the throughput of Raft4WS

clusters seems to have stagnated around 430 operations per second. Albeit allowing clients to connect to followers, hence sharing the load of processing their requests, by propagating them to the quorum and answering back to clients, ZooKeeper suffers more from failed followers, as the clients connected to a failed follower will need to connect to another server in the quorum to invoke subsequent requests. On the contrary, the performance of Raft4WS increases in a similar failure scenario, as the leader needs to contact a smaller number of followers and the clients will only need to reconnect when the leader fails. The failure of the leader causes an additional aggravation of the performance, as in Raft4WS it leads to clients reconnecting to the new leader, and in ZooKeeper, the service becomes unavailable until the new leader has been elected, and only then the clients will be able to reconnect to a server in the cluster.

Chapter 5

Case study

Smart Grids (SG) have been introducing a paradigm change in electric power system with the objective of enhancing the integration of renewables and promoting the generalized participation of different entities. Unprecedented research initiatives have been established with the purpose of addressing the architectural and technological aspects of power, information and communications systems [IEEE Guide for SG Interoperability, 2011; NIST SG Interoperability Standards, 2010], but important challenges in interoperability, reliability, and scalability need to be addressed before the Smart Grid vision can be fulfilled. The sheer scale of the electric grid and the criticality of the communication among its subsystems for proper management, demands a scalable and reliable coordination framework able to work in an heterogeneous and dynamic environment. The SG concept includes different visions and strategies that allow the modernization of the electric industry in order to ensure high levels of adaptability, scalability, security, economy, self-healing, robustness and protection in highly dynamic systems [Ipakchi and Albuyeh, 2009].

SG embody the future of the power grid because of the associated benefits, such as the reduction of carbon emissions and fuel costs, transmission losses, increased reliability to power failures, and deferral of investments, among others. Distributed Energy Resources (DER) are becoming widespread in power grids, in different segments of the power system, and require monitoring and control schemes to allow their enhanced participation in both market and system services.

The role of Information and Communication Technologies (ICT) in SG is gaining importance since it represents the underlying support infrastructure that allows the necessary information exchange towards the integration of different participants while supporting a diversified set of applications and services. As such, the need to provide full interoperability between diverse

current and future energy and non-energy systems, along with seamless discovery and configuration of a large variety of networked devices, ranging from the resource constrained sensing devices to servers in data centers, requires a suitable ICT system, which an implementation-agnostic Service Oriented Architecture. To achieve the complete integration of the various systems that compose the SG, a general ICT solution will need, among others, to: achieve the necessary interoperability between largely disparate devices; be scalable, in order to cope with the continuously increasing number of devices on the grid; and be highly reliable to support the operational requirements introduced by the SG.

Peer-to-peer communication and coordination protocols based on gossiping have been proposed to address scalability and reliability issues in SG, namely for secondary and tertiary control on a microgrid [De Brabandere et al., 2007], or for disseminating load shedding notification or aggregating metering data using the Automated Metering Infrastructure (AMI) [Krkoleva et al., 2011a]. However, reconciling such protocols with existing systems and paving the way for their long term maintenance and evolution is an open problem. On the other hand, Web Services in general, and the Devices Profile for Web Services (DPWS) [DPWS] in particular, can have an important role in SG [Karnouskos, 2012], as it provides several of the required features for an Energy SOA, by supporting dynamic, adaptive and auto-configurable architectures, and by embracing the heterogeneity on this environment, thus achieving full interoperability with energy systems based on the main standards and models [Schmutzler et al., 2011], as well as with other systems. However, the DPWS stack has only very limited support for peer-to-peer communication, assuming central coordination components. We thus propose to use a framework that provides gossip based dissemination and coordination as well as service replication built on top of Web Services, more precisely on DPWS, within the SG context, as it reconciles the reliability and scalability of Peer-to-Peer systems, with the industrial standard interoperability of Web Services. This framework allows taking full advantage of existing standards, including current devices, while paving the way for evolving to a decentralized peer-to-peer service architecture with replication capabilities that can be tuned according to each scenario's requirements.

5.1 Proposal

In order to evaluate qualitatively both services that comprise the proposed framework, we consider a simplified architecture of a Smart Grid (SG) focusing on the communications infrastructure depicted in Fig. 5.1. Briefly,

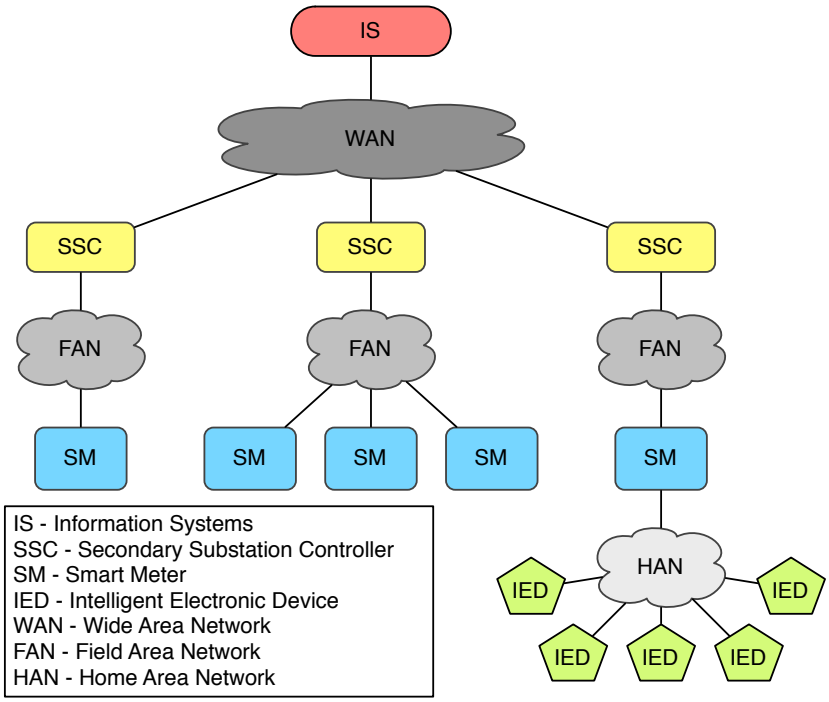


Figure 5.1: Overview of a simplified Smart Grid architecture.

the Information Systems (IS) of the utility are the main point for controlling and monitoring the entire grid, by retrieving data and issuing commands to other devices in the grid, such as Secondary Substation Controllers (SSC), normally connected to a Wide Area Network (WAN). SSC are installed in electric distribution transformers, and are equipped with sensors and actuators for monitoring the grid's conditions while enabling remote control. As previously mentioned, SSC interact with the IS, normally to report metrics or anomalies on the grid, and with Smart Meters (SM), connected to the same Field Area Network (FAN), to notify them on tariff changes or service perturbations. Smart Meters interface with customers, as well as with their appliances or Intelligent Electronic Devices (IED) through the Home Area Network (HAN), to convey relevant information such as metering and maintenance warnings.

Different types of data and scenarios inside a SG have different requirements, namely in terms of maximum allowed communications latency, and service availability. Protective relaying, status monitoring, and substation SCADA communications endure latency values as high as a few milliseconds to seconds or even minutes, but the loss of messages of these types is not tolerated due to their criticality to the SG operation [Li et al., 2012; Rua

et al., 2011]. Gossip protocols can be of particular importance in such settings, which are stricter in terms of message delivery assurance compared to message latency, as the message delivery assurance of these protocols largely outweighs the overhead of the additional traffic. Consensus protocols allow the usage of replication to allow services that are essential to the functioning of the SG to remain available, even in the case that some of its instances fail.

Our proposal to address the scalability and reliability challenges raised by the heterogeneity of the components of the SG, and its complex nature, is to use a Web Services framework for gossip-based dissemination and consensus-based fault tolerance. The inherent scalability and reliability of gossip protocols allows the usage of SOAP-over-UDP even if reliable delivery is desired, since it is much less resource consuming than a full-fledged HTTP binding over TCP. Moreover, by assuming the Web Services infrastructure based on the Devices Profile for Web Services (DPWS), we take advantage of each gossiped unit of data being a SOAP envelope, of the self-documenting nature of services through WSDL, and of useful base protocols and standards such as WS-Discovery and WS-Policy. Since DPWS does not offer any fault tolerance capability, it is necessary to resort to consensus protocols to ensure service availability and correctness. Hence, the proposed framework builds upon DPWS to promote interoperability among largely heterogeneous devices, from top of the range mainframes to small IED running completely different operating systems, and it is composed by three services, gossip, peer and consensus. The Gossip Service relies on gossip for disseminating messages, whereas the Peer Service provides information on the services and devices that are currently on the network. This information can then be used to build and enforce logical overlays on top of the SG's components, in order to guarantee communication among all of them. Gossip Service instances rely on this service to obtain the list of targets for disseminating messages. The Consensus Service is based on the Raft protocol, which incorporates the distributed state machine approach allowing the replication of commands to several instances of a service, in order to increase its availability.

5.2 Application scenarios

The usage of the proposed framework is illustrated in three specific scenarios: propagation of simple information, retrieval of distributed metrics, and propagation of important configurations. The first scenario focuses on the scalable dissemination capabilities of the framework, whereas the second one demonstrates its data aggregation capabilities, and the third focuses on the replication capabilities of the framework.

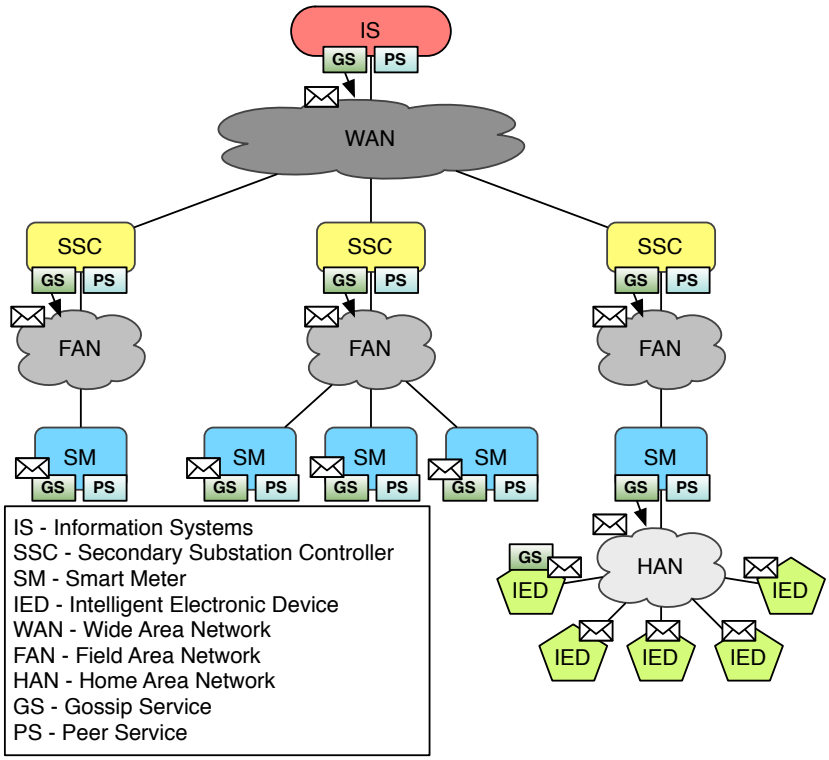


Figure 5.2: Overview of message dissemination using the proposed framework in a simplified Smart Grid architecture.

5.2.1 Propagation of simple information

On the first scenario, portrayed in Figure 5.2, assuming a dynamic tariff, where energy overproduction can lead to significant reduction of prices, these variations must be advertised to all the clients in order to adapt energy consumption accordingly, thus stabilizing the network by better matching the demand to the supply of energy.

The usage of dynamic price tariff schemes through AMI systems allows utilities to take advantage of operational scenarios to shape the participation of customers, by setting more, or less, attractive tariffs to them, while guaranteeing a stable and normal operation of the power system [Morgan et al., 2009]. The AMI comprises two-way communication between the utility's systems and SM, allowing the conveyance of information in both directions. These tariff modifications will then flow from the utility's IS to all the customers through their own SM or even through some other IED. We will focus on how these communications can occur using our framework in such a scenario.

When a utility decides to set lower price tariff through its IS, this information is then encapsulated in a push gossip message which is disseminated to the target SSC retrieved from the Peer Service deployed at the IS node. The Gossip Service instance of the targets, upon reception of the message, decrements the value of rounds r and retransmits the message to the target nodes that its Peer Service instance proposes, which could be other SSC, reachable through the WAN, or SM, reachable through the FAN to which the sending SSC is connected. When a SM receives the message, the Gossip Service instance behaves in a similar fashion to the one in the SSC, *i.e.*, it retransmits it to targets provided by the Peer Service instance. This instance can be located at the Smart Meter or at any other reachable node. The targets can vary from other SM, connected to the same FAN, to IED connected to the same HAN, that can range from controllable appliances to Renewable Energy Sources (RES). IED can then adapt their operating mode according to the received information of the tariff scheme. For instance, by analyzing the price reduction and the corresponding period, HVAC can increase its consumption to better suit the consumer's temperature preferences, while dishwashers and washing machines can anticipate their washing cycles, among other possibilities. In parallel with the retransmission of the message to the designated targets, the Gossip Service instance of the SM can present the notification on tariffs reduction in a local display or forward it to some other device, as configured by the customer, like a smartphone or a tablet.

5.2.2 Retrieval of distributed metrics

On the second scenario, in order to achieve better future power production planning, each consumer's SM can announce the energy requirements of the connected IED for a specific time frame, and this information will then be aggregated from level to level until reaching the utility's IS, as depicted in Figure 5.3.

The proposed framework is then used to collect metrics from different points of the SG in order to plan power production according to the announced energy requirements. For instance, charging of electric vehicles, and the usage of high power consumption appliances, such as dishwashers, are configured to occur during nighttime, when tariffs are usually lower. Periodically, the central IS invokes the Pull Aggregation operation on SSC, which, in their turn, invoke the same operation on the target SM designated by their Peer Service. A SM, upon reception of such a request, propagates the same request to the IED pointed by the Peer Service in the HAN. Each of these IED, if configured to perform some scheduled task, will respond with the energy requirements to execute these tasks, their duration, and the time

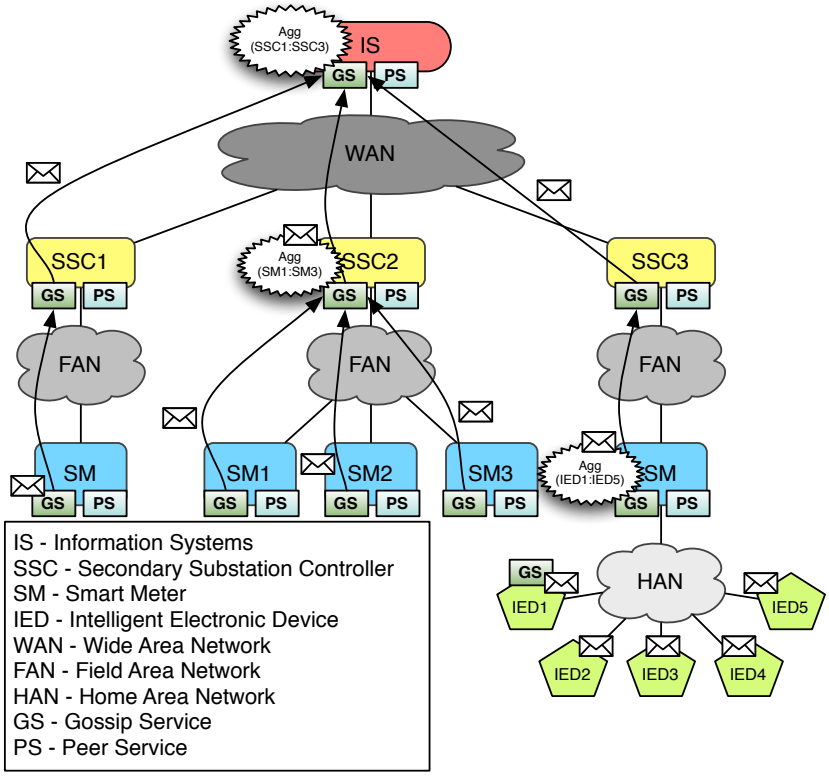


Figure 5.3: Overview of message aggregation using the proposed framework in a simplified Smart Grid architecture.

by which they should be finished. The SM will then aggregate this information for the entire household, after waiting for responses from IED until a certain number of responses arrives or a certain timeout elapses, according to configured preferences. The aggregated information will combine all the power requirements pointed by the IED for the three 8 hour time periods which divide the day. For simplification purposes, we will consider that all the energy needs for each of these periods will be simply added in order to produce the aggregate information at the SM. Each SSC will then receive the aggregate responses from the previously contacted SM, and again, having waited according to configured preferences, will aggregate those responses in a single message sent back to the IS. The IS will then process this message and assess what are the announced energy requirements and plan the energy generation according to the demand for the next periods.

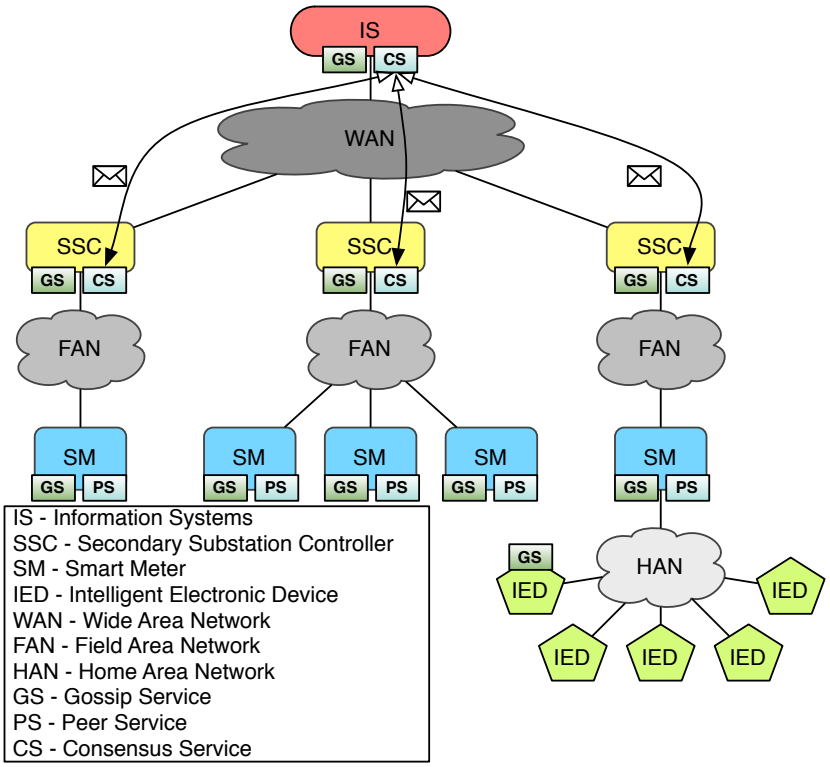


Figure 5.4: Overview of command replication using the proposed framework in a simplified Smart Grid architecture.

5.2.3 Propagation of important configurations

The third scenario, illustrated in Figure 5.4, assuming that the IS sets a new value for the grid's operating voltage, which must be replicated to all the SSC in order to stabilize the operation of the SG.

This scenario exercises the service correctness abilities of the proposed framework, through the usage of the Consensus Service to ensure that all its instances apply the same command, which in this case is the setting of a new value for the operating voltage of the SG, which could be necessary, for instance, to compensate some disturbance probably due to a generator malfunction or failure. When this parameter is modified in the Consensus Service instance of the IS, which can be considered as the leader for simplification purposes, the corresponding command is processed, by appending it to the log, and then forwarded to its known followers, namely the instances of the Consensus Service hosted by the SSC. According to the used Consensus Protocol, the followers process the command and respond to the leader with the status of that processing. When the leader receives a majority of

successful responses from the followers, the new command is applied to the state machine and this fact is conveyed to the followers, so they can replicate this modification, in order to achieve service consistency. In this way, all the correct Consensus Service instances ensure the configuration of a new value for the voltage parameter through the described communications which are illustrated in Figure 5.4.

5.3 Related work

The smart metering infrastructure typically is a hierarchical architecture composed by three layers [Karnouskos, 2012], which are, from the base to the top of the hierarchy: the meter layer that corresponds to the actual meters or to Intelligent Electronic Devices (IED) with similar capabilities, *i.e.* can measure the amount of energy that is consumed or produced; the concentrator layer, where concentrators aggregate measurements from IED on the meter layer and send it to the metering data system (MDS); the metering data management layer, where the analysis and management of the gathered metering data is performed, for instance for billing, forecasting and other purposes.

Various approaches on smart metering target a 15 minute resolution, which can be considered a metering of high density [Karnouskos, 2012], while others are more ambitious and assume that every electric meter generates a message every second and transmits it to the connected feeder or distribution substation [Aggarwal et al., 2010]. This level of detailed information and interaction allows better monitoring of all the online assets and better control of power generation, as it can be adjusted to better suit the demand.

It has also been shown that DPWS is suitable for smart meters communication [Karnouskos and Izmaylova, 2009], but for a large amount of devices, in the region of some thousands, an hierarchically structured communication does not cope well with the generated traffic [Karnouskos et al., 2011]. Either way, a DPWS-enabled smart meter device would prove its usefulness by rendering its functionalities accessible through Web Services, which can be dynamically discovered by interested client applications or devices. This allows heterogeneous devices, mostly still to be deployed in the future, to interact with the smart meter to better adapt their energy consumption and integrate the future smart grid without major impacts in terms of software maintenance.

Another approach to Smart Grid's problems based on Web Services is WS-SCADA [Chen et al., 2006], which addresses integration needs of clients, applications, utilities and market participants, by accommodating all their

information needs of all participants and adapting to dynamic changes at both system and business level. The proposed open, flexible and scalable infrastructure for information integration includes two Web Services protocol stacks, for a control center and a substation, and both share with DPWS a lot of similarities in their composition, as well as its Achilles' heel, communication disruption due to network and service failures. For instance, the WS-Discovery protocol provides Plug-and-Play features for IED allowing substations to locate them and their services. WS-Eventing can be used by substations to receive notifications from IED, and by control centers to notify substations on control messages or to be notified on status information and real–time operation data of substations, such as voltage, current, breaker status, and phasor measurements.

Gossip protocols can be used to disseminate important information in the Smart Grid (SG), for instance, load shedding notifications, or metering data aggregation [Krkoleva et al., 2011a,b]. However, limiting epidemic dissemination to a single pairwise interaction per node in each cycle leads to large dissemination times. The more traditional approach, where each gossiping node contacts various of its neighbors in parallel would prove to be useful in the majority of the SG scenarios.

Gossip protocols can also be used to implement secondary and tertiary control in microgrids, with the aim of improving power quality and optimizing generation costs [De Brabandere et al., 2007; Vanthournout et al., 2005]. A gossip aggregation protocol can be used to calculate the average of voltage and frequency deviations measured at Distributed Energy Resources (DER) units, which can then be added to the reference active and reactive power in order to stabilize the network, by decreasing the deviations. To optimize distributed generation costs, each DER unit periodically contacts a random neighbor in order to harmonize their marginal cost functions. Such scenarios could benefit from more advanced aggregation strategies, in order to decrease the number of communication interactions between the DER units.

5.4 Summary

The implementation of Smart Grids is highly supported by Information and Communication Technologies, integrating different computing and networking elements which are present in the multitude of systems composing current and future power grids.

In this chapter, we have shown how this challenge can be addressed by instantiating the proposed flexible service framework. In short, it builds first on gossip based communication variants providing probabilistic message delivery

guarantees as well as proactive reliability, and improves service availability and correctness through consensus-based fault-tolerance. Secondly, we leverage the Devices Profile for Web Services, which allows communications based on Web Services between resource constrained devices and mainframes, automatic detection of the devices present on the network and easy integration of new devices.

The main contribution of our work is showing how simple peer-to-peer primitives, for gossiping and membership management, when properly integrated in a service framework, are a powerful foundation for different communication and coordination applications. The same goes with the integration of a generic consensus protocol in the same service framework, which further increases the achieved reliability, beyond communications, into the service execution. This flexibility is key in infrastructures such as Smart Grids, whose current deployments are expected to last for a long period of time and to evolve as new technologies are integrated and new requirements are addressed.

Chapter 6

Conclusions

The Devices Profile for Web Services (DPWS) specification has enabled resource constrained devices to interact through Web Services, but does not provide any built-in fault tolerance mechanism. While using this specification, the occurrence of a stoppage in lengthy operations with many interveners, namely due to communication or service faults, means they cannot be resumed. This type of events can be catastrophical if critical steps, such as urgent alerts or commands, are interrupted. To prevent its occurrence, scalable lightweight coordination and replication protocols were built upon DPWS, in order to ensure service dependability. However, the usage of these protocols must ensure that the modularity and interoperability of the existing services is not modified in any way. Thus, the expansion of Service-Oriented Computing to largely heterogeneous environments, namely comprising less powerful devices, introduced some issues in terms of service integration, such as reliable communication and service correctness. Information dissemination in the context of Service-Oriented Architectures involving large numbers of connected devices poses a set of challenges that are not adequately met with traditional approaches.

To address them, we propose the usage of gossiping at an architectural level instead of either relegating the information dissemination problem to black box middleware or coping with the limitations of heavyweight coordination protocols and their assumptions of buffering and transactional logs for reliability. Gossiping has several advantages in this context, as a variety of protocols can be achieved with minimal complexity and provide strong guarantees of reliable and atomic dissemination. These include both one-to-many and many-to-many dissemination, as well as many-to-one aggregation queries. In contrast to previous approaches [Renesse et al., 2003], our proposal integrates seamlessly in a DPWS environment, being compatible with existing devices. By implementing the proposed architecture on the WS4D

JMEDS stack, we show that the performance of a one-to-many operation using gossip improves on bare SOAP-over-UDP, included in DPWS, both on latency and fault tolerance, while offering additional flexibility and resilience.

With the aim of increasing the reliability of lightweight middleware architectures based on DPWS, a Web Services framework was proposed, which comprises WS-Gossip, to enable efficient and reliable message dissemination, and Raft4WS, a consensus service that guarantees fault tolerance at the service level. This framework was evaluated quantitatively with micro-benchmarks and qualitatively with a case study in order to illustrate its utility. WS-Gossip has proved to be very resource efficient and reliable, even in the presence of communication faults, when compared with other communication protocols, achieving in most scenarios a smaller time for disseminating messages, and it additionally provides aggregation capabilities which reduces the number of exchanged messages. WS-Gossip also provides useful message exchange patterns such as many-to-one or one-to-may on heterogeneous environments, where Web Services specifications for reliable messaging are limited to point-to-point communication and show interoperability issues.

From the presented results, we can conclude that Raft4WS is suitable for small scale Web Services scenarios, with a limited number of clients and servers, performing better than Apache ZooKeeper on these settings by achieving better throughput and lower latency. Moreover, the usage of DPWS provides a confident basis for the adoption of such a system to provide consensus in scenarios with largely heterogeneous devices, where churn could be overcome by the Ad-Hoc mode of WS-Discovery. In scenarios with more concurrency, *i.e.* more clients and requests, Apache ZooKeeper clusters have the upper hand, expressed in the superior throughput which increases with the size of the cluster and seems to have not yet reached the saturation point, contrarily to Raft4WS. Furthermore, it is possible to conclude that the overhead introduced by the usage of the proposed framework, when compared to existing middleware platforms, is reasonable and this impact is made up by the improvements arising from the usage of Service-Oriented Architectures, namely in terms of modularity.

In comparison to WS-PushGossip and WS-Membership, which build upon the WS-Coordination standard, the proposed services are better suited for devices and their inherent limitations, due to their scarce resource consumption and their ability to leverage standards included in DPWS. While WS-PushGossip only provides *eager push* dissemination, the proposed framework can provide additional variants of gossip dissemination. Moreover, WS-Membership implies that specific monitoring agents are capable of assessing the availability of services with specific mechanisms, whereas the presented framework relies on WS-Discovery for the same purpose.

6.1 Future work

In the future of the proposed framework stand some prospects regarding its usefulness, namely in the increasing adoption of the Internet of Things in several environments, from houses to enterprises and industries, which has been enabled by the Devices Profile for Web Services (DPWS). To integrate heterogeneous devices in houses, the usage of DPWS has already been exemplified in the Smart Grid [Karnouskos, 2012; Karnouskos and Izmaylova, 2009], and the proposed framework can leverage that experience to provide reliable services and communication, as proposed in the case study. The integration of devices in enterprise and manufacturing systems introduces a greater level of automation and cost reduction, as communication becomes more transparent and direct without the need of translation between different system layers, as previously provided by third party solutions and device drivers, which now have become obsolete [Karnouskos et al., 2009; Spiess et al., 2009].

This research has not focused on the security aspects of the proposed framework in any way. Hence, a complete analysis to the security from both the service and communication perspectives of the framework should be performed to eliminate any safety issues regarding its usage in real scenarios.

Further developments on both WS-Gossip and Raft4WS can be made, for instance, by using the Efficient XML Interchange (EXI) format, which can improve the performance of this framework, through the reduction of both the size of the exchanged messages, consequently decreasing used bandwidth, and the time expended to process messages [Jammes et al., 2011].

Bibliography

IEEE Standards Coordinating Committee 21. IEEE Guide for Smart Grid Interoperability of Energy Technology and Information Technology Operation with the Electric Power System (EPS), End-Use Applications, and Loads. *IEEE Std 2030-2011*, pages 1–126, Sept 2011. (Cited on page 95.)

Amit Aggarwal, Swathi Kunta, and Pramode K. Verma. A Proposed Communications Infrastructure for the Smart Grid. In *Proceedings of the first IEEE PES Conference on Innovative Smart Grid Technologies (ISGT 2010)*, pages 1–5, Jan 2010. (Cited on page 103.)

Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag Berlin Heidelberg, 2004. (Cited on pages 14, 15, 16, and 57.)

Hussein Badakhchani. Introduction to RosettaNet. `http://web.archive.org/web/20070807130129/http://dev2dev.bea.com/pub/a/2004/12/RosettaNet.html`, 6 December 2004. (Cited on pages 15 and 57.)

Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999. (Cited on pages 32, 37, and 67.)

BTP. OASIS Business Transaction Protocol (BTP) Committee Specification 1.0. `http://www.oasis-open.org/committees/download.php/1184/2002-06-03.BTP_cttee_spec_1.0.pdf`, 3 June 2002. (Cited on page 16.)

Luis Felipe Cabrera and Chris Kurt. *Web Services Architecture and Its Specifications: Essentials for Understanding WS-**. Microsoft Press, Richmond, USA, 9 February 2005. (Cited on page 20.)

Nuno A. Carvalho, João Bordalo, Filipe Campos, and José Pereira. Experimental Evaluation of Distributed Middleware with a Virtualized Java Environment. *Proceedings of the 6th Workshop on Middleware for Service*

Oriented Computing (MW4SOC 2011), pages 1–7, Oct 2011. (Cited on page 77.)

Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 173–186, February 1999. (Cited on page 29.)

Hua Chai, Honglei Zhang, Wenbing Zhao, P. Michael Melliar-Smith, and Louise E. Moser. Toward Trustworthy Coordination of Web Services Business Activities. *IEEE Transactions on Services Computing*, 6(2):276–288, April 2013. (Cited on page 29.)

Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996. (Cited on pages 39, 41, 43, and 46.)

Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, July 1996. (Cited on pages 39, 43, and 44.)

Qizhi Chen, Hamada Ghenniwa, and Weiming Shen. Web-Services Infrastructure for Information Integration in Power Systems. In *IEEE Power Engineering Society (PES) General Meeting, 2006*, 2006. (Cited on page 103.)

Cover Pages WS-CAF. Web Services Composite Application Framework (WS-CAF) for Transaction Coordination. `http://xml.coverpages.org/ni2003-07-29-a.html`, 29 July 2003. (Cited on page 21.)

Karel De Brabandere, Koen Vanthournout, Johan Driesen, Geert Deconinck, and Ronnie Belmans. Control of Microgrids. In *IEEE Power Engineering Society (PES) General Meeting, 2007*, pages 1–7. IEEE, June 2007. (Cited on pages 96 and 104.)

Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12. ACM, 1987. (Cited on page 32.)

DPWS. 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. (Cited on pages 9 and 96.)

Thomas Erl. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall/PearsonPTR, 2004. (Cited on page 26.)

Thomas Erl. *Service-Oriented Architecture: Concepts, Technology and Design*. Prentice Hall/PearsonPTR, August 2005. (Cited on page 26.)

Patrick T. Eugster, Rachid Guerraoui, Sidath B. Handurukande, Petr Kouznetsov, and Anne-Marie Kermarrec. Lightweight Probabilistic Broadcast. *ACM Transactions on Computer Systems*, 21(4):341–374, November 2003. (Cited on page 37.)

Patrick T. Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulié. Epidemic Information Dissemination in Distributed Systems. *IEEE Computer*, 37(5):60–67, May 2004. (Cited on pages 31, 33, 35, 36, 37, 59, and 79.)

Donald F. Ferguson, Tony Storey, Brad Lovering, and John Shewchuk. Secure, Reliable, Transacted Web Services:Architecture and Composition. `https://msdn.microsoft.com/en-us/library/ms996535.aspx`, September 2003. (Cited on pages 24 and 25.)

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with One Faulty Process. *Journal of the Association for Computing Machinery*, 32(2):374–382, April 1985. (Cited on pages 39 and 50.)

David Gregorczyk. WS-Eventing SOAP-over-UDP Multicast Extension. *Proceedings of the 9th IEEE International Conference on Web Services (ICWS 2011)*, pages 660 – 665, July 2011. (Cited on page 57.)

Rachid Guerraoui and Michel Raynal. A Generic Framework for Indulgent Consensus. *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003)*, pages 88–95, May 2003. (Cited on pages 39, 42, 43, and 45.)

Rachid Guerraoui and André Schiper. The Generic Consensus Service. *IEEE Transactions on Software Engineering*, 27(1):29–41, Jan 2001. (Cited on pages 3, 39, 40, 41, and 59.)

Weiping He. Recovery in Web Service Applications. *Proceedings of the 2004 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'04)*, pages 25–28, March 2004. (Cited on pages 2, 30, and 58.)

Ali Ipakchi and Farrokh Albuyeh. Grid of the Future. *IEEE Power and Energy Magazine*, 7(2):52–62, 2009. (Cited on page 95.)

François Jammes, Antoine Mensch, and Harm Smit. Real-time performance Web Services using EXI. *Proceedings of the 37th Annual Conference on IEEE Industrial Electronics Society (IECON 2011)*, November 2011. (Cited on page 109.)

Deepal Jayasinghe. FAWS for SOAP-based web services. `http://www.ibm.com/developerworks/webservices/library/ws-faws/`, 31 Jan 2005. (Cited on pages 23 and 28.)

Márk Jelasity, Wojtek Kowalczyk, and Maarten van Steen. Newscast Computing. Technical Report IR-CS-006, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands, November 2003. (Cited on pages 34 and 68.)

Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware (Middleware '04)*, pages 79–98, October 2004. (Cited on pages 37 and 38.)

Matjaz B. Juric, Ramesh Loganathan, Poornachandra Sarang, and Frank Jennings. *SOA Approach to Integration - XML, Web services, ESB, and BPEL in real-world SOA projects.* Packt Publishing, November 2007. (Cited on pages 13 and 26.)

Stamatis Karnouskos. Asset monitoring in the service-oriented internet of things empowered smartgrid. *Service Oriented Computing and Applications*, 6(3):207–214, 2012. (Cited on pages 56, 96, 103, and 109.)

Stamatis Karnouskos and Anastasia Izmaylova. Simulation of Web Service Enabled Smart Meters in an Event-based Infrastructure. *Proceedings of the 7th IEEE International Conference on Industrial Informatics (INDIN 2009)*, pages 125 – 130, June 2009. (Cited on pages 56, 103, and 109.)

Stamatis Karnouskos, Thomas Bangemann, and Christian Diedrich. Integration of Legacy Devices in the Future SOA-based Factory. *Proceedings of the 13th IFAC Symposium on Information Control Problems in Manufacturing (INCOM'2009)*, June 2009. (Cited on page 109.)

Stamatis Karnouskos, Per Goncalves da Silva, and Dejan Ilic. Assessment of High-performance Smart Metering for the Web Service Enabled Smart

Grid. *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering (ICPE'11)*, pages 133–144, August 2011. (Cited on pages 56 and 103.)

Richard M. Karp, Christian Schindelhauer, Scott J. Shenker, and Berthold Vöcking. Randomized Rumor Spreading. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 565–574. IEEE, 2000. (Cited on pages 31, 32, 34, 37, 63, and 67.)

Anne-Marie Kermarrec and Maarten van Steen. Gossiping in Distributed Systems. *ACM SIGOPS Operating Systems Review*, 41(5):2–7, 2007a. (Cited on pages 31, 59, and 60.)

Anne-Marie Kermarrec and Maarten van Steen. Gossip-Based Computer Networking. *ACM SIGOPS Operating Systems Review*, 41(5), October 2007b. (Cited on page 61.)

Setrag Khoshafian. *Service Oriented Enterprises*. Auerbach Publications, 2007. (Cited on page 23.)

Boris Koldehofe. Simple gossiping with balls and bins. In *Proceedings of the 6th International Conference on Principles of Distributed Systems (OPODIS'02)*, pages 109–118, December 2002. (Cited on page 35.)

Boris Koldehofe. Buffer management in probabilistic peer-to-peer communication protocols. In *Proceedings of the 22nd Symposium on Reliable and Distributed Systems (SRDS 2003)*. IEEE, IEEE, October 2003. (Cited on page 65.)

Aleksandra Krkoleva, Vesna Borozan, Aris L. Dimeas, and Nikos D. Hatziargyriou. Requirements for Implementing Gossip Based Schemes for Information Dissemination in Future Power Systems. In *Proceedings of the 2nd IEEE PES International Conference and Exhibition on Innovative Smart Grid Technologies (ISGT Europe 2011)*, pages 1–7. IEEE, 2011a. (Cited on pages 96 and 104.)

Aleksandra Krkoleva, Vesna Borozan, Aris L. Dimeas, and Nikos D. Hatziargyriou. Gossip Based Message Dissemination Schemes in Future Power Systems. In *Proceedings of the 16th International Conference on Intelligent System Application to Power Systems (ISAP'11)*, pages 1–6. IEEE, 2011b. (Cited on page 104.)

Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978. (Cited on page 54.)

Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998. (Cited on pages 48 and 56.)

Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, 1 Nov. 2001. (Cited on pages 49, 50, 51, and 59.)

Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, July 1982. (Cited on pages 23 and 28.)

Depeng Li, Zeyar Aung, John R Williams, and Abel Sanchez. Efficient Authentication Scheme for Data Aggregation in Smart Grid with Fault Tolerance and Fault Diagnosis. In *Proceedings of the third IEEE PES Conference on Innovative Smart Grid Technologies (ISGT 2012)*, pages 1–8. IEEE, Jan 2012. (Cited on page 97.)

Wei Li, Jiang He, Qingkai Ma, I-Ling Yen, Farokh Bastani, and Raymond Paul. A Framework to Support Survivable Web Services. *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Jan 2005. (Cited on pages 23 and 29.)

Deron Liang, Chen-Liang Fang, Chyouhwa Chen, and Fengyi Lin. Fault tolerant web service. *Proceedings of the Tenth Asia-Pacific Software Engineering Conference (APSEC'03)*, pages 310–319, Dec. 2003. (Cited on pages 23 and 28.)

Barbara H. Liskov and James Cowling. Viewstamped replication revisited. `http://18.7.29.232/handle/1721.1/71763`, Jan 2012. (Cited on pages 53 and 59.)

Mark Little and Thomas Freund. A comparison of Web services transaction protocols. `http://www.ibm.com/developerworks/webservices/library/ws-comproto/`, 7 October 2003. (Cited on page 13.)

Mark Little and Jim Webber. Introducing WS-CAF - More than just transactions. `http://webservices.sys-con.com/read/39936.htm`, 1 December 2003. (Cited on page 57.)

Nik Looker, Malcolm Munro, and Jie Xu. Increasing Web Service Dependability Through Consensus Voting. *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*, Jan 2005. (Cited on pages 24 and 30.)

Ben Margolis and Joseph Sharpe. *SOA for the Business Developer: Concepts, BPEL, and SCA*. MC Press, Lewisville, USA, 15 May 2007. (Cited on pages 13, 14, and 23.)

James McGovern, Oliver Sims, Ashish Jain, and Mark Little. *Enterprise Service Oriented Architectures: Concepts, Challenges, Recommendations*. Springer, Berlin, Germany, 1st edition, 28 April 2006. (Cited on pages 16 and 57.)

Michael G. Merideth, Arun Iyengar, Thomas Mikalsen, Stefan Tai, Isabelle Rouvellou, and Priya Narasimhan. Thema: Byzantine-Fault-Tolerant Middleware for Web-Service Applications. *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005)*, pages 131–140, 26-28 Oct. 2005. (Cited on pages 23, 28, and 29.)

Minha. Minha: Middleware Testing Platform. `http://www.minha.pt/`. (Cited on page 77.)

Nader Mohamed and Jameela Al-Jaroodi. A survey on service-oriented middleware for wireless sensor networks. *Service Oriented Computing and Applications*, 5(2):71–85, Jan 2011. (Cited on page 56.)

Geert Monsieur, Monique Snoeck, and Wilfried Lemahieu. Coordinated Web Services Orchestration. *Proceedings of the IEEE International Conference on Web Services (ICWS 2007)*, pages 775–783, July 2007. (Cited on page 57.)

M. Granger Morgan, Jay Apt, Lester Lave, Marija D. Ilic, Marvin A. Sirbu, and Jon M. Peha. The many meanings of "Smart Grid". *repository.cmu.edu*, Jan 2009. (Cited on page 99.)

Achour Mostefaoui, Michel Raynal, and Corentin Travers. Crash-resilient Time-free Eventual Leadership. *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS 2004)*, pages 208–217, Oct. 2004. (Cited on page 39.)

National Institute of Standards and Technology. NIST Framework and Roadmap for Smart Grid Interoperability Standards, Release

1.0, 2010. http://www.nist.gov/public_affairs/releases/upload/smartgrid_interoperability_final.pdf. (Cited on page 95.)

Eric Newcomer and Greg Lomow. *Understanding SOA with Web Services*. Addison Wesley Professional, 14 December 2004. (Cited on page 19.)

Brian M. Oki and Barbara H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. *Proceedings of the 7th annual ACM Symposium on Principles of Distributed Computing (PODC '88)*, Jan 1988. (Cited on pages 53 and 59.)

Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. *ramcloud.stanford.edu*, 2013. (Cited on pages 53, 56, 59, and 69.)

Johannes Osrael, Lorenz Froihofer, and Karl M. Göschka. On the Need for Dependability Research on Service Oriented Systems. *Fast Abstract Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN'07)*, 25-28 June 2007a. (Cited on pages 23, 24, 27, and 58.)

Johannes Osrael, Lorenz Froihofer, Martin Weghofer, and Karl M. Göschka. Axis2-based Replication Middleware for Web Services. *Proceedings of the IEEE International Conference on Web Services (ICWS 2007)*, pages 591–598, July 2007b. (Cited on pages 2, 23, 27, 28, and 58.)

Sajeeva L. Pallemulle and Kenneth J. Goldman. Byzantine Fault-Tolerant Web Services for n-Tier and Service Oriented Architectures. In *Proceedings of the 28th International Conference on Distributed Computing Systems (ICDCS '08)*, pages 260–268, June 2008. (Cited on pages 23 and 29.)

José Pereira and Rui Oliveira. The Mutable Consensus Protocol. *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS 2004)*, pages 218–227, 18-20 Oct. 2004. (Cited on pages xiii, 2, 39, 45, 46, 47, and 48.)

José Pereira, Luís Rodrigues, Maria José Monteiro, Rui Oliveira, and Anne-Marie Kermarrec. Neem: network-friendly epidemic multicast. *Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS 2003)*, pages 15–24, 6-18 Oct. 2003. (Cited on page 35.)

José Pereira, Rui Oliveira, and Luís Rodrigues. Efficient Epidemic Multicast in Heterogeneous Networks. In *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, volume 4278/2006, pages 1520–1529, October 2006. (Cited on pages 34, 35, and 37.)

Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21(2):164–206, May 2003. (Cited on page 107.)

David Rua, L. F. Moura Pereira, Nuno Gil, and João Abel Peças Lopes. Impact of multi-Microgrid Communication systems in islanded operation. In *2nd IEEE PES Internatinal Conference and Exhibition on Innovative Smart Grid Technologies (ISGT Europe),*, pages 1–6. IEEE, 2011. (Cited on page 97.)

Jorge Salas, Francisco Pérez-Sorrosal, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. WS-Replication: A Framework for Highly Available Web Services. In *Proceedings of the 15th International Conference on World Wide Web (WWW '06)*, pages 357–366. ACM, 23-26 May 2006. (Cited on pages 2, 23, 24, 27, 28, and 58.)

André Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10:149–157, March 1997. (Cited on page 46.)

Jens Schmutzler, Sven Gröning, and Christian Wietfeld. Management of distributed energy resources in IEC 61850 using web services on devices. In *Proceedings of the second IEEE International Conference on Smart Grid Communications (SmartGridComm'11)*, pages 315–320. IEEE, 2011. (Cited on page 96.)

Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4): 299–319, 1990. (Cited on page 48.)

Dale Skeen. Nonblocking commit protocols. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, pages 133–142. ACM, 1981. (Cited on page 42.)

SOAP-over-UDP. SOAP-over-UDP Version 1.1 OASIS Standard. `http://docs.oasis-open.org/ws-dd/soapoverudp/1.1/os/wsdd-soapoverudp-1.1-spec-os.html`, 01 July 2009. (Cited on page 9.)

Patrik Spiess, Stamatis Karnouskos, Dominique Guinard, Domnic Savio, Oliver Baecker, Luciana Moreira Sá de Souza, and Vlad Trifa. SOA-Based Integration of the Internet of Things in Enterprise Services. *Proceedings of the 7th IEEE International Conference on Web Services (ICWS 2009)*, pages 968 – 975, July 2009. (Cited on page 109.)

Koen Vanthournout, Karel De Brabandere, Edwin Haesen, Jeroen Van den Keybus, Geert Deconinck, and Ronnie Belmans. Agora: Distributed Tertiary Control of Distributed Resources. In *Proceedings of the 15th Power Systems Computation Conference (PSCC-15)*, page 7, 2005. (Cited on page 104.)

Werner Vogels and Chris Re. WS-Membership - Failure Management in a Web-Services World. *International World Wide Web Conference (WWW) Alternate Paper Tracks*, Jan 2003. (Cited on pages 2, 30, 31, 39, and 58.)

Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. Cyclon: Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005. (Cited on page 37.)

Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-ReliableMessaging, and More*. Prentice Hall PTR, March 2005. (Cited on page 23.)

Matthias Wiesmann, Xavier Défago, and André Schiper. Group communication based on standard interfaces. *Second IEEE International Symposium on Network Computing and Applications (NCA 2003)*, pages 140–147, 16-18 April 2003. (Cited on pages 39 and 40.)

Wiki ebXML. Wikipedia - ebXML. `http://en.wikipedia.org/wiki/Ebxml`. (Cited on page 15.)

Wiki Overlay network. Wikipedia - Overlay network. `http://en.wikipedia.org/wiki/Overlay_network`. (Cited on page 37.)

Wiki Paxos. Wikipedia - Paxos algorithm. `http://en.wikipedia.org/wiki/Paxos_algorithm`. (Cited on pages 49, 52, and 53.)

Wiki WS-CAF. Wikipedia - WS-CAF. `http://en.wikipedia.org/wiki/WS-CAF`. (Cited on page 21.)

Wiki xCBL. Wikipedia - xCBL. `http://en.wikipedia.org/wiki/Xcbl`. (Cited on page 15.)

WS-AT. Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.2 OASIS Standard. `http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec-os.pdf`, 2 February 2009. (Cited on pages 11, 18, and 57.)

WS-BA. Web Services Business Activity (WS-BusinessActivity) Version 1.2 OASIS Standard. `http://docs.oasis-open.org/ws-tx/wstx-wsba-1.2-spec-os.pdf`, 2 February 2009. (Cited on pages 20 and 57.)

WS-C. Web Services Coordination (WS-Coordination) Version 1.2 OASIS Standard. `http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os.pdf`, 2 February 2009. (Cited on pages 11, 17, and 57.)

WS-DD. 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. (Cited on page 9.)

WS-E. Web Services Eventing (WS-Eventing) W3C Member Submission. `http://www.w3.org/Submission/WS-Eventing/`, 15 March 2006. (Cited on pages 9 and 11.)

WS-ME. Web Services Metadata Exchange 1.1 (WS-MetadataExchange) W3C Member Submission. `http://www.w3.org/Submission/2008/SUBM-WS-MetadataExchange-20080813/`, 13 August 2008. (Cited on page 9.)

WS-P. Web Services Policy (WS-Policy) 1.5 - Framework W3C Recommendation. `http://www.w3.org/TR/ws-policy/`, 04 September 2007. (Cited on page 9.)

WS-R. Web Services Reliability (WS-Reliability) 1.1 OASIS Standard. `http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf`, 15 November 2004. (Cited on pages 26 and 57.)

WS-RM. Web Services Reliable Messaging (WS-ReliableMessaging) Version 1.2 OASIS Standard. `http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.2-spec-os.pdf`, 2 February 2009. (Cited on pages 24 and 57.)

WS4D. Web Services for Devices (WS4D). `http://www.ws4d.org/`. (Cited on pages 10, 77, and 85.)

Wenbing Zhao. BFT-WS: A Byzantine Fault Tolerance Framework for Web Services. In *Proceedings of the 11th International IEEE Enterprise Distributed Object Computing Conference (EDOC 2007) Workshop*, pages 89–96, Oct 2007. (Cited on pages 23 and 29.)