Coordenação de Serviços Web heterogéneos com tolerância a faltas

Filipe Campos, Miguel Matos e José Pereira

HASLab - High-Assurance Software Laboratory
INESC TEC e Universidade do Minho
Campus de Gualtar
4710-057 Braga, Portugal
{fcampos,miguelmatos,jop}@di.uminho.pt

Resumo A norma Devices Profile for Web Services (DPWS) permite a descoberta, a configuração e a interoperabilidade de dispositivos heterogéneos ligados em rede com grande disparidade em termos de capacidade de processamento, desde pequenos eletrodomésticos inteligentes ou controladores em máquinas industriais, até servidores em centros de dados. Os mecanismos de notificação de eventos e de configuração automática previstos pelo DPWS são especialmente adequados a cenários Machine-to-Machine (M2M) devido à sua simplicidade e flexibilidade. No entanto, a escalabilidade em cenários com um elevado número de nós, nomeadamente, em grandes infraestruturas, é limitada. A coerência dos dados armazenados e manipulados pelos mecanismos de autodescoberta também não é adequada a aplicações críticas. Neste artigo, abordamos este problema com uma proposta assente na norma DPWS com os conceitos de difusão epidémica e de consenso, mostrando que é possível compor serviços adequados a diferentes aplicações assegurando garantias de tolerância a faltas e maior escalabilidade.

1 Introdução

As arquiteturas orientadas a serviços (SOA) são um importante pilar para a computação empresarial e há um crescente interesse na sua utilização em dispositivos localizados nos mais variados ambientes, desde máquinas altamente especializadas para produção industrial até aos eletrodomésticos inteligentes. Isto sucede devido à melhoria na relação custo-benefício para equipar dispositivos com capacidades de processamento e de comunicação consideráveis, mas também devido à flexibilidade na interoperabilidade, combinação e composição de serviços.

Neste contexto, a possibilidade de expor as capacidades de dispositivos como serviços é atraente, mas levanta problemas como a disseminação escalável e tolerante a faltas [3], a manutenção da filiação no serviço [27] ou a cadência de transmissão face a destinatários de capacidade limitada [24].

Existem duas abordagens para a disseminação eficiente e fiável de informação na computação orientada a serviços. A primeira encapsula a lógica de disseminação em middleware, através de tecnologias como Enterprise Service Bus

(ESB), Java Messaging Service (JMS) ou Extensible Messaging and Presence Protocol (XMPP) [2,12], introduzindo uma dependência de uma determinada pilha de software e, frequentemente, requer servidores centralizados, ainda que tornados redundantes através de replicação. Para além disso, este middleware não está normalmente disponível para a variedade de dispositivos que têm que ser suportados. Finalmente, ao fornecer capacidades de comunicação como uma caixa negra, os padrões de troca de mensagens são bastante limitados. A segunda abordagem consiste na disseminação de informação ao nível do serviço, através de especificações que podem ser combinadas para expor diferentes padrões de troca de mensagens com variados níveis de *QoS*. Um exemplo desta abordagem é a norma WS-Eventing, que apesar de não suportar *brokers*, incorpora de raiz um mecanismo flexível de filtragem de mensagens, favorecendo implementações eficientes e disseminação de um para muitos. É de notar, contudo, que ambas as abordagens enfatizam a comunicação de um para muitos através de uma infraestrutura centralizada. A tolerância a faltas do servidor central assenta na replicação, enquanto a fiabilidade ponto a ponto e a atomicidade depende do facto de os participantes terem memória suficiente e armazenamento persistente para o registo transacional. Resumidamente, este pressuposto de uma infraestrutura pesada e centralizada é partilhada pela maioria das soluções existentes para a disseminação de informação na computação orientada a serviços.

Devices Profile for Web Services (DPWS) define um conjunto de protocolos e especificações que os dispositivos com recursos limitados devem implementar para interagir transparentemente através de Serviços Web. Cada dispositivo fornece a funcionalidade básica e expõe um ou mais serviços que fornecem as funcionalidades específicas do dispositivo. Entre outras, esta norma inclui especificações como WS-Eventing, SOAP-over-UDP, que permite a utilização de UDP como transporte para as mensagens SOAP e de comunicação multi-ponto, e WS-Discovery, WS-MetadataExchange e WS-Policy, que possibilitam a descoberta dinâmica de dispositivos na rede e respetiva caracterização ao nível dos serviços e recursos. Ao focar-se em sistemas de pequena escala, como domótica, a especificação DPWS apresenta limitações de escalabilidade, impedindo o seu uso com um grande número de componentes. Desde logo, a utilização de WS-Eventing impõe um fardo ao publicador de informação, ao obrigá-lo a notificar todos os subscritores e, por não suportar mecanismos de coordenação transacional, pode levar a estados do sistema incorretos com múltiplos intervenientes.

A alternativa seria recorrer a middleware de coordenação desenvolvido para infraestruturas de grande dimensão como o Apache ZooKeeper [13], que contudo requer Java SE 1.6, não sendo compatível com o sistema operativo Android ou Java Micro Edition, que são normalmente disponibilizados em plataformas com poucos recursos. Por outro lado, a norma DPWS fornece uma plataforma flexível para a implementação de serviços e disseminação de eventos, mas sem tolerância a faltas. De facto, embora seja possível adaptar protocolos de coordenação existentes para Serviços Web, como WS-Coordination e WS-AtomicTransaction, os modelos de faltas consideram infraestruturas empresariais com muitos recursos.

Portanto, um protocolo de coordenação leve e escalável, que se encaixe nas suposições gerais da norma DPWS, é necessário.

Neste artigo, propomos uma infraestrutura para a coordenação de serviços com tolerância a faltas que se baseia em especificações atuais para Serviços Web bem como em protocolos epidémicos e protocolos de consenso.

Os protocolos epidémicos são uma abordagem escalável e frugal para a disseminação de mensagens. Apesar de se poder utilizar um middleware já existente (ex. NeEM[1]), teríamos muitas das limitações já apontadas à primeira abordagem. Alternativamente, e em linha com a segunda abordagem, fornecemos serviços que podem ser combinados para reproduzir uma variedade de interações epidémicas, permitindo que qualquer arquitetura orientada a serviços utilize este tipo de protocolos para múltiplas aplicações de disseminação de informação.

O serviço de consenso fornecido na infraestrutura apresentada oferece a funcionalidade do serviço de coordenação ZooKeeper, não fornecida pela norma DPWS. Como a notificação de eventos se encontra incorporada em DPWS, esta traduz-se na capacidade de replicar dados de forma tolerante a faltas. A nossa abordagem para este serviço assentou na implementação do protocolo de consenso Raft [20] sobre DPWS de forma a fornecer aos serviços replicados a capacidade de tolerar falhas de comunicação bem como de servidores.

O resto do artigo apresenta a seguinte estrutura. A Secção 2 contextualiza e motiva o trabalho. Na Secção 3 é apresentada a arquitetura geral, discutindo-se a disseminação na Secção 3.1 e a coordenação na Secção 3.2. A Secção 4 descreve a avaliação conduzida, a Secção 5 discute o trabalho relacionado e por fim a Secção 6 conclui o artigo.

2 Contexto

A coordenação de serviços envolvendo disseminação de informação preocupa-se com dois desafios principais: a) como disseminar informação de forma eficiente para todos os alvos e como manter as listas de alvos e b) que garantias são dadas de que a informação é realmente disseminada na presença de faltas.

No remanescente desta secção, serão descritas as normas para Serviços Web focados nestes desafios, bem como os protocolos epidémicos e de consenso.

Desde os anos 90 que a comunicação confiável tem sido vista como a solução para estes desafios, e assim, várias tecnologias de filas de mensagens transacionais têm sido utilizadas, tais como WebSphereMQ da IBM e MSMQ da Microsoft, para além de tecnologias de publicação/subscrição, como TIBCO Rendezvous. Num esforço para interligar estas diferentes tecnologias, a API Java Message Service (JMS) foi definida, normalizando a utilização de ambas entre aplicações diversas. Algumas destas tecnologias foram adaptadas aos serviços Web, mas, devido à dependência de protocolos proprietários, a interoperabilidade requer *gateways* que façam a tradução entre ambientes diferentes.

[1] http://neem.sf.net

Com a emergência dos Serviços Web como solução de integração preferida para sistemas distribuídos, WS-ReliableMessaging (WS-RM) é a norma atualmente adotada para realizar trocas confiáveis de mensagens em aplicações distribuídas na presença de falhas de software, de máquinas ou de rede. Assegurando a interoperabilidade entre serviços heterogéneos, em termos da confiabilidade da comunicação, o desenvolvimento de serviços baseados nesta norma é simplificado, minimizando o número de erros na lógica de negócio [11].

O termo coordenação refere-se por vezes a um tipo de orquestração definido pela norma WS-Coordination, que especifica uma infraestrutura extensível para gestão de contexto para a coordenação de ações em aplicações distribuídas. Esta coordenação é alcançada através de protocolos que estendem esta norma para, por exemplo, alcançar um acordo no resultado de transações distribuídas, como é o caso de WS-AtomicTransaction (WS-AT), para transações atómicas e de curta duração, e de WS-BusinessActivity (WS-BA), para atividades de negócio de longa duração. A norma WS-AT fornece uma adaptação do mecanismo clássico *2PC* para serviços Web, apesar de se considerar que não funciona bem com esta tecnologia [19]. No entanto, mostra-se adequado para interações curtas entre serviços que se encontrem próximos e que necessitem de resultados coerentes para uma transação.

Outras normas, como WS-Eventing, WS-AT ou WS-BA, podem utilizar WS-RM para garantir comunicação confiável entre os intervenientes. Apesar de ser capaz de garantir a entrega de mensagens ponto a ponto de forma confiável, a sua utilização é ineficiente e provoca uma sobrecarga no emissor no caso de haver muitos destinatários ou muitos erros de comunicação. Para garantir a entrega atómica para todos os destinos, a norma WS-RM teria que recorrer a WS-AT ou um protocolo de coordenação semelhante, o que aumentaria o consumo de recursos tanto de processamento como de comunicação. Assim sendo, não é capaz de lidar por si própria com falhas para oferecer um serviço replicado, enfatizando a necessidade de um algoritmo capaz e eficiente para tal cenário.

Nas comunicações entre computadores, a comunicação epidémica descreve o processo em que um participante, que pretende disseminar alguma informação, seleciona aleatoriamente um pequeno conjunto de outros participantes e lhes envia essa informação. Cada um destes destinos, repete este procedimento, sendo também frequentemente conhecida por rumor (*gossip*) por se assemelhar à propagação de boatos [10]. Um aspeto interessante destes protocolos é que não necessitam de um mecanismo reativo a falhas, nomeadamente, de armazenamento, confirmação ou retransmissão de mensagens que constituem a maior parte da complexidade dos protocolos de comunicação mais comuns. Em vez disso, a confiabilidade é assegurada pela redundância e aleatoriedade inerentes aos protocolos, permitindo lidar com falhas tanto de processos como de comunicação. A probabilidade esperada para a entrega de uma mensagem para cada destino e para todos os destinos pode ser derivada diretamente dos parâmetros f, número de alvos selecionados localmente por cada processo, e r, número máximo de vezes que uma mensagem deve ser reencaminhada antes de ser descartada. Ajustando f e r consoante as dimensões do sistema e número de faltas esperadas, a entrega

da mensagem a qualquer número médio desejado de destinos, ou até atómica, pode ser garantida [10]. A chave para a escalabilidade é que o valor do parâmetro f é na pior das hipóteses logaritmicamente proporcional à dimensão do sistema.

O consenso é uma abstração do problema em que todos os processos de um sistema distribuído devem acordar no mesmo valor, apesar de terem iniciado com opiniões diferentes e independentemente de alguns falharem [23].

O Raft [20] é um protocolo de consenso que segue a abordagem da máquina de estados replicada. Nele são dissociados elementos chave dos algoritmos de consenso, como a eleição de líder, a replicação do registo persistente e a coerência, enquanto simplifica a concretização através da redução do número de estados possíveis. O Raft suporta modificações à constituição do grupo, mantendo-se em normal funcionamento durante tais transições. Comparativamente com o ZooKeeper, que também utiliza a noção de líder, o Raft é um protocolo mais simples pois requer a implementação de um menor número de operações distintas, e minimiza a funcionalidade das réplicas. Por exemplo, as entradas de registo persistente fluem num único sentido, do líder para as réplicas, enquanto no ZooKeeper, estas entradas fluem em ambos os sentidos.

3 Framework

A proposta para ultrapassar os desafios identificados é uma infraestrutura composta pelos serviços WS-Gossip e Raft4WS, cuja arquitetura está ilustrada na Figura 1, assentando sobre os protocolos contidos na norma DPWS implementada pela Java Multi Edition DPWS Stack (JMEDS) do projeto Web Services for Devices (WS4D).

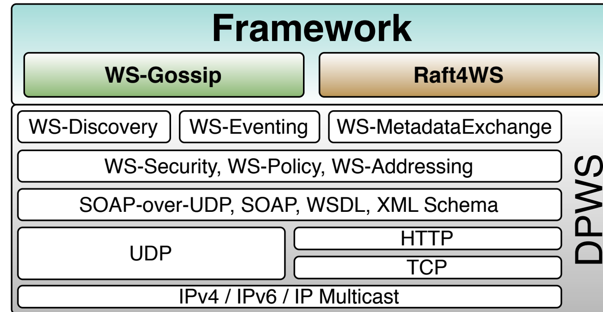


Figura 1. Visão global da arquitetura proposta no contexto da norma DPWS.

3.1 WS-Gossip

Nesta secção é descrito o serviço WS-Gossip, incluindo as operações disponibilizadas e a forma como se integra com aplicações DPWS existentes.

Quando instalado num dispositivo, o WS-Gossip analisa os serviços nele alojados, criando um serviço sombra para cada um deles. Tanto o serviço original como o seu serviço sombra são anunciados aos clientes que podem utilizar cada um deles independentemente. Um cliente com capacidades WS-Gossip pode examinar as anotações de políticas em ambos os serviços e descobrir que estão relacionados. Sendo possível enviar pedidos diretamente para o serviço original, é assegurada a compatibilidade com clientes que não tenham capacidades WS-Gossip.

Assumindo uma operação *one-way* ou de notificação e *push*, descreveremos o funcionamento básico do WS-Gossip. A disseminação é iniciada quando um cliente envia uma mensagem SOAP para uma operação do serviço sombra. Após a sua receção, a mensagem é inspecionada para determinar se contém um cabeçalho WS-Gossip. Caso não contenha tal cabeçalho, este é construído utilizando os parâmetros predefinidos, como por exemplo, a variante ou o número de alvos (f). A disseminação é então iniciada através da inclusão deste cabeçalho na mensagem, que é reencaminhada para os f alvos obtidos do serviço de filiação e para o serviço original. Quando um participante recebe uma mensagem do WS-Gossip, decrementa o contador presente na mensagem enviando-a, de seguida, aos seus alvos para que a disseminação prossiga.

A utilização em operações *one-way* ou de notificação é tratada como foi descrito anteriormente. Em operações de pedido-resposta ou de *call-back*, a mensagem recebida é propagada e posteriormente todas as respostas recebidas são devolvidas até ao iniciador da disseminação. Isto requer que o seu endereço seja incluído no cabeçalho SOAP, nomeadamente no campo **ReplyTo** de WS-Addressing, para além do identificador da mensagem utilizado para a deteção de duplicados. A alternativa é utilizar um filtro que pode omitir ou agregar respostas de acordo com a regra especificada no parâmetro **Filter** do cabeçalho WS-Gossip aquando do início da disseminação.

O serviço sombra oferece as seguintes operações para além das que replica do serviço original:

Push Alternativa à invocação direta da operação replicada, permitindo também o envio de um conjunto de mensagens numa única interação.

PushIds Informa o alvo dos identificadores das mensagens que o dispositivo invocador possui. O alvo deve depois invocar a operação **Fetch** indicando os identificadores das mensagens que pretende receber.

Pull Retorna as mensagens recebidas e armazenadas durante um período de tempo especificado como parâmetro.

PullIds Variante da operação anterior, mas que retorna os identificadores em vez das mensagens, que podem ser obtidas através da invocação da operação **Fetch** indicando os seus identificadores.

Fetch Devolve as mensagens armazenadas cujos identificadores foram passados como parâmetro.

O WS-Gossip suporta várias variantes gossip que podem ser *lazy* ou *eager*, relativamente à prontidão de envio, e *pull* ou *push*, relativamente à preferência na receção ou envio destas mensagens, sendo realizadas através da composição

das operações descritas anteriormente. Nomeadamente, *lazy push* resulta da invocação de **PushIds**, em vez de **Push** como em *eager push*, aguardando por invocações a **Fetch** para enviar as mensagens identificadas. *Eager pull* é obtida pela invocação periódica da operação **Pull**. A variante *lazy pull* resulta da invocação periódica de **PullIds** para obter identificadores de mensagens, invocando **Fetch** para obter as mensagens pretendidas. A variante selecionada para a disseminação de determinada mensagem depende da configuração do serviço.

3.2 Raft4WS

Nesta secção descreve-se o serviço Raft4WS, que possui duas entidades: o Servidor, que aloja uma instância do serviço, e o Cliente, que efetua pedidos aos Servidores.

De acordo com o protocolo Raft, um Servidor pode assumir um de três estados diferentes, *seguidor*, *candidato* ou *líder*, e arranca sempre como seguidor. O serviço Raft4WS inclui 3 operações que todas as suas instâncias fornecem e que têm correspondência com os RPCs definidos no protocolo Raft: **InsertCommand**, que é invocada pelos clientes para inserir novos comandos no registo persistente replicado pelo grupo; **AppendEntries**, invocada pelo líder como *heartbeat*, quando não houver novas entradas no registo, ou para replicar tais entradas nos seguidores; **RequestVote**, que é invocada pelos candidatos para angariar os votos de outros Servidores. O objeto representativo de cada estado inclui as tarefas associadas e implementa uma interface comum para tratamento de disparos do temporizador, caso o servidor não seja contactado, e de invocações às operações do serviço.

A principal tarefa do Cliente é a deteção de dispositivos Raft, através da escuta de mensagens multicast WS-Discovery, ou através da procura ativa por tais dispositivos. O primeiro dispositivo Raft detetado será considerado como líder pelo Cliente, tornando-se o alvo das suas invocações. O Cliente invoca a operação **InsertCommand** com o comando e os parâmetros, bem como um identificador único como parâmetros. Se o Servidor contactado for o líder atual, a resposta conterá o resultado da criação da entrada no registo correspondente ao pedido do Cliente e da sua aplicação na máquina de estados replicada. Caso contrário, o Servidor responde ao Cliente indicando o insucesso do pedido e o endereço do líder atual. Neste caso, o Cliente pode utilizar este endereço para reenviar o pedido ao líder, para que este seja processado, bem como os seguintes.

4 Avaliação de Desempenho

4.1 Raft4WS

Avaliámos a nossa implementação do protocolo Raft, Raft4WS, comparando-a com o Apache ZooKeeper 3.4.5 em cenários com 1, 3 e 5 servidores, para responder aos pedidos de 1 a 25 clientes. Cada cenário foi testado 5 vezes num grupo de 6 máquinas, correspondendo os resultados apresentados à sua média.

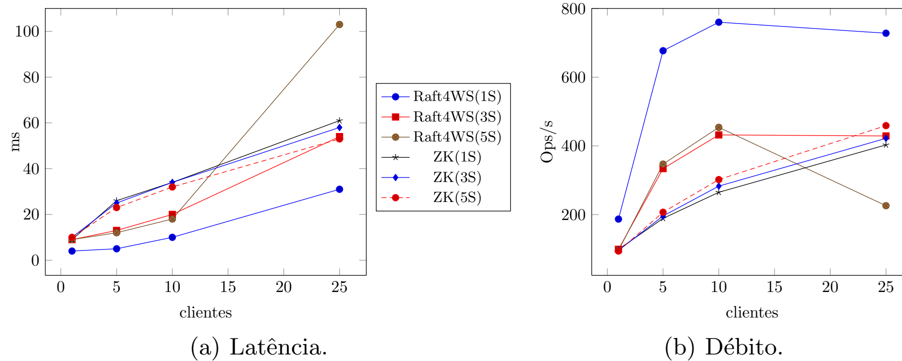
Cada máquina tinha a seguinte configuração: 64-bit Ubuntu 12.04.4 Linux, Intel^{R} Core^{TM} i3-2100, 3.10GHz, 8GB RAM, 64-bit Java^{TM} SE 1.6.0_27. Uma máquina foi utilizada exclusivamente para correr o gestor e todos os clientes, enquanto as restantes máquinas executaram um único servidor Raft4WS ou ZooKeeper. Cada cliente executou 120 iterações sem qualquer intervalo, invocando a inserção de um novo comando no servidor líder, no caso do Raft4WS, ou num servidor qualquer, no caso do ZooKeeper, sendo este o comportamento do cliente fornecido. Para a medição da latência, as primeiras e as últimas 10 iterações foram descartadas para minimizar o efeito do arranque e da paragem do sistema, apesar de encobrir o estabelecimento de ligações TCP, enquanto o débito considerou as 120 iterações.

A execução de cada teste ao Raft4WS começa com a inicialização do gestor e clientes na mesma máquina e dos servidores nas suas máquinas. Quando o gestor deteta que todos os clientes e servidores iniciaram, seleciona o líder, informando todos os clientes sobre este facto e sobre o número de iterações. O gestor informa cada servidor sobre o respetivo estado, os membros do grupo e o parâmetro Raft *election timeout*, que toma um valor aleatório entre 150 e 300ms para cada um deles. Em seguida, o gestor notifica-os para arrancarem o seu funcionamento normal, e subscreve a notificação do final das iterações nos clientes, indicando-lhes que comecem a execução das mesmas. Os clientes executam o número indicado de iterações, invocando a operação **InsertCommand** do líder em cada uma delas. No final, os clientes notificam o gestor, que aguarda o contacto de todos os clientes, para os informar, bem como aos servidores, do ficheiro em que devem colocar as estatísticas.

O procedimento para o ZooKeeper é exatamente o mesmo, com a exceção da interação entre o gestor e os servidores, visto que o código destes não foi alterado. Os parâmetros de configuração utilizados foram os seguintes: *tickTime* de 2 segundos, que é o intervalo entre *heartbeats*; *initLimit* de 5 *ticks*, que é o período de tempo que um servidor tem para se ligar ao líder; *syncLimit* de 2 *ticks*, que é o atraso máximo do estado de um servidor comparativamente com o líder do quórum.

Os efeitos de 2 servidores falhados também foram avaliados no grupo de 5 servidores, uma vez que ultrapassando este limite o serviço deve ficar indisponível para garantir a sua correção. Com este propósito, o gestor foi configurado para causar a falha de um seguidor meio segundo após o início do teste, seguida por outra falha após igual período de tempo, que pode ser de outro seguidor (2F) ou do líder (1F1L). O cenário base será identificado por 0E quando comparado com estes cenários com falhas. Para comparar de forma mais justa o comportamento de conexão do cliente ao grupo, após perda de ligação, reproduzimos o comportamento dos clientes ZooKeeper no cliente Raft4WS, em que este deve aguardar por um período de tempo aleatório até ao máximo de 1 segundo, antes de se tentar ligar a outro servidor.

A Figura 2(a) ilustra que a latência de todos os cenários cresce linearmente com o número de clientes, excetuando Raft4WS(5S), onde esta parece aumentar exponencialmente de 10 para 25 clientes, mostrando possivelmente sinais de



(a) Latência. (b) Débito.

Figura 2. Raft4WS vs. ZooKeeper.

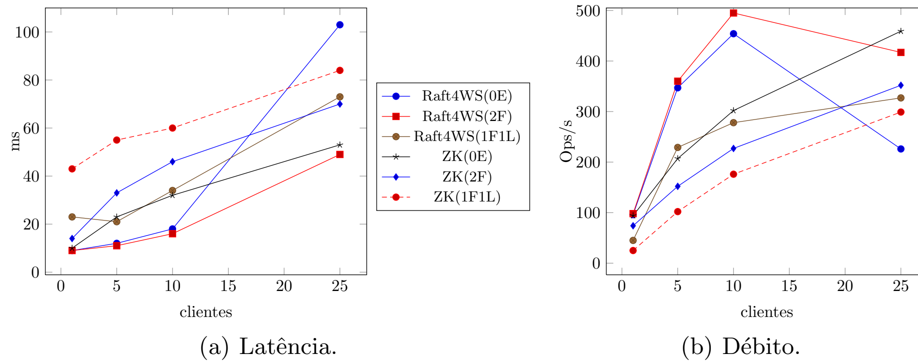
saturação dos recursos do grupo de servidores. A latência de Raft4WS(1S) corresponde ao serviço não replicado, e é previsivelmente inferior à dos restantes cenários, para qualquer número de clientes. É importante referir que a latência nos vários cenários ZooKeeper diminui à medida que o número de servidores aumenta. Outro facto importante é que a latência de Raft4WS para 3 e 5 servidores é bastante semelhante e até inferior ao cenário ZooKeeper correspondente, com a exceção do caso já mencionado para Raft4WS(5S).

A Figura 2(b) apresenta os valores de débito dos servidores nos vários cenários, sugerindo uma relação próxima com os valores de latência da Figura 2(a). Relativamente aos cenários ZooKeeper, é possível observar que o débito aumenta com o número de servidores, suportando as suas conhecidas capacidades de paralelismo e alta disponibilidade, à medida que os pedidos dos clientes vão sendo distribuídos pelos vários servidores do grupo. Por outro lado, o protocolo Raft apenas permite que o líder responda a pedidos de clientes, afetando a sua escalabilidade com esta limitação, uma vez que o líder pode facilmente ficar sobrecarregado.

A influência de 2 servidores falhados no desempenho de um grupo com 5 servidores é ilustrada nas Figuras 3(a) e 3(b). A latência média de qualquer cenário ZooKeeper é sempre superior à do cenário Raft4WS correspondente, aumentando linearmente com o número de clientes em todos os cenários. A única exceção, tal como foi referido anteriormente, é o caso de Raft4WS(0E) com 25 clientes.

A falha de servidores ZooKeeper provoca uma degradação do desempenho, devido à necessidade de os clientes, ligados ao servidor que falhou, terem que se ligar a outro servidor. A latência aumenta entre 4 e 17ms, e o débito baixa entre 9 e 31 Ops/s em ZK(2F). Em ZK(1F1L), o desempenho sofre uma degradação adicional, ficando os valores de latência cerca de 30ms mais elevados e o débito 20 a 130 Ops/s abaixo do cenário base.

A falha de 2 seguidores melhora o desempenho do Raft4WS, reduzindo ligeiramente a latência e aumentando o débito entre 2 a 40 Ops/s, excetuando o caso de Raft4WS(0E) com 25 clientes. Este é também o único caso em que



(a) Latência.

(b) Débito.

Figura 3. Raft4WS vs. ZooKeeper com 5 servidores com falhas de servidores aos 500 e aos 1000ms.

o Raft4WS(1F1L) tem melhor desempenho. Nos restantes casos, este cenário introduz uma penalização de 9 a 16ms na latência e uma redução de 50 a 180 Ops/s no débito.

Para finalizar, o desempenho do Raft4WS é sempre melhor que a do ZooKeeper em condições semelhantes, excetuando Raft4WS(5S). Contudo, a Figura 2(b) mostra uma tendência de crescimento do débito do ZooKeeper para além dos 25 clientes, parecendo estagnar para o Raft4WS em torno das 430 Ops/s. Apesar de os clientes ZooKeeper se ligarem a qualquer servidor, este protocolo sofre mais que o Raft4WS com a falha de seguidores, pois os clientes ligados a estes terão que se ligar a outro servidor para invocar novos pedidos. Pelo contrário, o desempenho do Raft4WS melhora com estas falhas, sendo reduzido o número de mensagens que o líder tem que enviar aos seguidores restantes, e que os clientes apenas terão que se ligar a outro servidor no caso de o líder falhar. A falha do líder causa sempre um agravamento adicional do desempenho, uma vez que provoca a eleição de novo líder. Os clientes ligam-se ao novo líder no Raft4WS, enquanto, no ZooKeeper, o serviço fica indisponível até esta eleição terminar, podendo então os clientes criar a nova ligação.

4.2 WS-Gossip

Na avaliação do WS-Gossip e protocolos alternativos, foi utilizada a plataforma de testes middleware Minha [5,1], dado não ser possível executar no nosso cluster um número de nós suficiente para observar a globalidade das propriedades dos protocolos epidémicos. Esta plataforma virtualiza múltiplos dispositivos numa única JVM, ao mesmo tempo que simula um sistema real, permitindo também a injeção de falhas de rede e consequente avaliação de tolerância a faltas em vários cenários. Cada teste corresponde à simulação da execução de um determinado número de dispositivos numa LAN, utilizando Minha numa máquina com a seguinte configuração: 64-bit Ubuntu Server 10.04.4 Linux, dois processadores

AMD Opteron™ 6172 com 12 núcleos a 2.1GHz, 128 GB RAM, 64-bit Sun Microsystems Java SE 1.6.0_26. A avaliação consistiu na disseminação periódica de um evento contendo um valor numérico representando a temperatura medida por um produtor para um determinado número de consumidores, sendo cada teste controlado por um gestor. Foram avaliados os seguintes cenários:

WS-E WS-Eventing incluído em WS4D JMEDS, com transporte HTTP/TCP.

WS-RM Implementação simplificada de WS-ReliableMessaging, com transporte SOAP-over-UDP e garantia de entrega *AtLeastOnce*. Neste cenário, a receção de cada mensagem é imediatamente confirmada ao produtor, que aguarda 50ms, tempo de espera mínimo para a retransmissão em SOAP-over-UDP, antes de a reenviar.

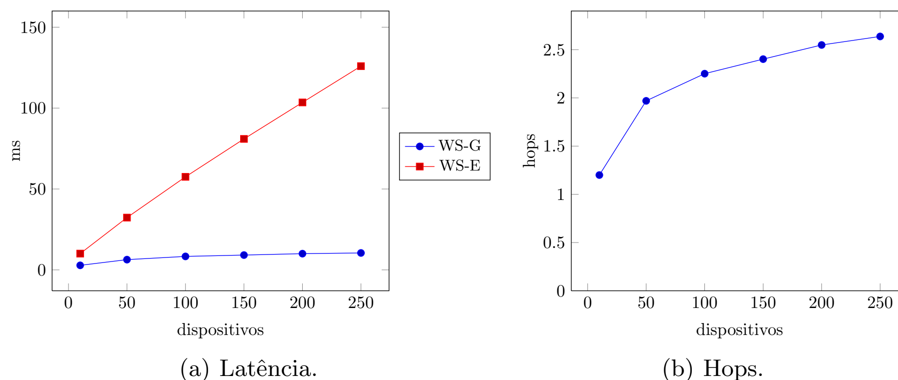
WS-G Variante *eager push* do WS-Gossip sobre SOAP-over-UDP.

AggWS-G Estende WS-G com agregação de mensagens, através de um filtro XSLT, que neste caso processava a média dos valores de entrada.

Cada teste começa com a inicialização do gestor, produtor e consumidores. Em WS-E, estes últimos subscrevem-se junto do produtor. Em WS-G e AggWS-G, o gestor informa cada participante sobre os seus vizinhos. O gestor informa o produtor sobre os consumidores em WS-RM, para que possa criar uma sequência de troca de mensagens confiável com cada um deles. Em todos os cenários, o gestor verifica se todos os dispositivos arrancaram corretamente antes de indicar ao produtor para começar a disseminação. O produtor começa a gerar eventos periodicamente, propagando-os pela rede, e terminada a disseminação, notifica o gestor. Em WS-RM, o produtor fecha também as sequências utilizadas para comunicar com os consumidores. O gestor indica a cada dispositivo em que ficheiro deve escrever as estatísticas.

Cada teste foi executado 5 vezes para cada cenário com um determinado número de consumidores, onde 120 eventos foram disseminados com um intervalo de 5s. Em todos os cenários, exceto WS-E, introduziu-se entre 0 e 20% de perdas de mensagens, para avaliar o seu efeito na degradação da confiabilidade e latência. A latência foi definida como o intervalo entre o instante da emissão de uma mensagem pelo produtor e a sua receção pelo consumidor, sendo a sua medição facilitada pelo facto de a execução de todos os dispositivos ocorrer numa única JVM. Em WS-G e AggWS-G, os valores usados para o parâmetro f foram calculados consoante [10], considerando o número de dispositivos, uma taxa esperada de erros de 5% e uma garantia de entrega de 99%, variando entre 8, para 10 dispositivos, e 11 para 250 dispositivos. Nestes mesmos cenários, o produtor é escolhido aleatoriamente entre todos os dispositivos pelo gestor, contrariamente aos restantes cenários em que é designado o primeiro dispositivo. Nos resultados apresentados de medição de latências, as primeiras e as últimas 10 iterações foram descartadas para minimizar o efeito da compilação JIT Java, apesar de encobrir o estabelecimento de ligações TCP em WS-E.

Na Figura 4(a), a latência de WS-E aumenta linearmente com o número de consumidores, de 10 para 125ms, enquanto em WS-G aumenta muito devagar, de 3 para 10ms. Isto justifica-se através da dispersão da carga de propagação de uma mensagem pelos dispositivos na rede, em vez de sobrecarregar um único

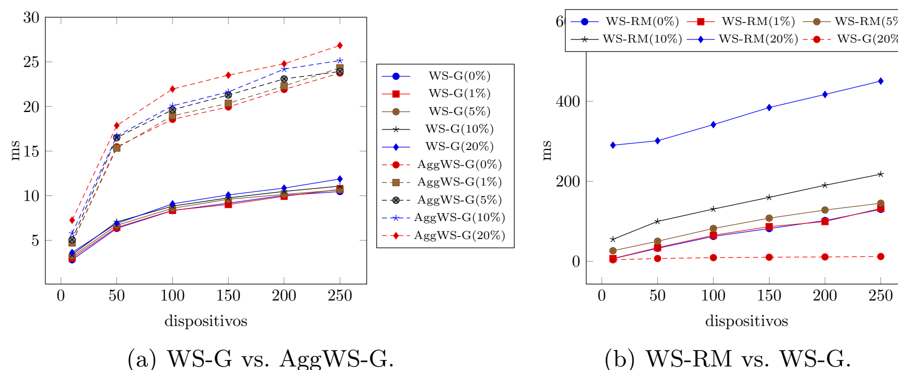


(a) Latência.

(b) Hops.

Figura 4. WS-E vs. WS-G.

dispositivo, como sucede com o produtor em WS-E. A Figura 4(b) apresenta o crescimento logarítmico da média de *hops* que a mensagem sofre desde a emissão à receção em WS-G, confirmando que escala de forma logarítmica com o tamanho do sistema.



(a) WS-G vs. AggWS-G.

(b) WS-RM vs. WS-G.

Figura 5. Comparação de WS-G com AggWS-G e WS-RM (Latência).

É possível observar a latência de ambos os cenários de gossip na Figura 5(a), que ilustra o *overhead* da utilização de agregação. Comparando ambos os cenários base, o acréscimo de 2 a 13ms na latência, com 10 e 250 dispositivos respetivamente, pode ser considerado um pequeno preço a pagar para a inclusão desta funcionalidade. Nos vários cenários, a introdução de perdas de mensagens aumenta um pouco a latência, variando entre 0,5 e 1,5ms, de 0% para 20% de perdas em WS-G, e entre 2,5 e 3,5ms, de 0% para 20% de perdas em AggWS-G.

A Figura 5(b) ilustra os efeitos da perda de mensagens em WS-RM e também permite compará-lo com o cenário WS-G com mais perdas. É possível observar que o desempenho de WS-RM se aproxima do de WS-G para poucos consumidores, quando não há perdas. A sua latência aumenta com o número de consumidores, de 6 até 130ms, afastando-se assim de WS-G, em que a latência nunca ultrapassa 12ms. Com 1 e 5% de perdas, a latência de WS-RM é bastante próxima de WS-RM(0%), apresentando um acréscimo na latência entre 0,3 e 3ms e entre 15 e 27ms, respetivamente. Na presença de 10% de perdas, a latência já apresenta um acréscimo entre 55 e 218ms. Com 20% de perdas, a latência começa em 290ms, para 10 consumidores, e termina em 450ms, para 250 consumidores. Este acréscimo elevado, comparativamente com os casos anteriores, demonstra o impacto da acumulação de inúmeras retransmissões resultantes da ocorrência de perdas.

A taxa de entrega de mensagens não é apresentada graficamente uma vez que é de 100% para WS-E e WS-RM, e que em WS-G e AggWS-G, é sempre superior a 99,9%, chegando a 100% com grande frequência, mesmo com 10% de mensagens perdidas, o que corresponde ao dobro da taxa esperada de erros de 5%.

Os resultados apresentados mostram as limitações de WS-Eventing na presença de muitos dispositivos. Como o produtor é obrigado a manter uma conexão TCP com cada subscritor, o consumo de recursos elevado é confirmado pelo aumento claro da latência com o número de subscritores. A nossa intenção em utilizar WS-ReliableMessaging sobre UDP era tolerar falhas de comunicação, sem o custo adicional do TCP, esperando um bom desempenho. Contudo, os resultados mostram que este se degrada bastante à medida que as perdas aumentam, sendo o caso extremo de 20% de perdas anormalmente penalizador. Da comparação de WS-Gossip com outros protocolos, fica claro que o seu desempenho é pouco penalizado pelo aumento no número de destinos, demonstrando a sua capacidade de escalar, bem como pela introdução de falhas, mostrando a sua elevada confiabilidade.

Em cenários com n dispositivos e em que um único produtor tem de os contactar a todos, este terá que enviar n mensagens por evento, enquanto os participantes no protocolo epidémico enviam f mensagens, permitindo assim dispersar a carga da disseminação por toda a rede, resultando numa menor carga para o produtor nos casos em que $n > f$.

5 Trabalho Relacionado

A computação orientada a serviços já provou ser um paradigma muito adaptável, mesmo em ambientes com escassos recursos, tal como as redes de sensores sem fios [18]. A especificação Devices Profile for Web Services (DPWS) é a chave para a implementação do paradigma *Internet of Things*, tornando-se essencial para a Smart Grid [14], possibilitando a interação de entidades como contadores inteligentes e sistemas de gestão de energia das companhias elétricas. Esta interação permite uma melhor monitorização de todos os aparelhos e um melhor

controlo da produção elétrica, podendo ajustá-la de acordo com o consumo medido. Outra abordagem aos problemas da Smart Grid, baseada em serviços Web, é o WS-SCADA [8], que acomoda as necessidades de informação de todos os participantes e reage a mudanças tanto ao nível do sistema como do negócio, através de duas pilhas de protocolos que partilham com DPWS a utilização de WS-Discovery, para funcionalidade *plug-and-play* entre participantes, e de WS-Eventing, para a notificação de eventos. Os protocolos epidémicos podem ser utilizados para a disseminação de informação importante, sejam notificações de cortes de energia ou dados agregados de contagem [15], mas também para melhorar a qualidade de energia e otimizar os custos de geração em Microgrids [9]. Comparando com WS-PushGossip [4], que apenas oferece *eager push* gossip, e com WS-Membership [28], que monitoriza a atividade de serviços Web usando gossip para trocar atualizações, o WS-Gossip é mais adequado a dispositivos e às suas limitações, uma vez que oferece diversas variantes de gossip, e não se baseia em WS-Coordination, alavancando também os protocolos incluídos em DPWS. Para além disso, a utilização de agentes de monitorização específicos para determinar a disponibilidade dos serviços em WS-Membership é uma desvantagem comparativamente com a utilização de WS-Discovery pelo WS-Gossip para o mesmo propósito.

A replicação ativa de serviços Web é fornecida por WS-Replication [26], que envia as invocações recebidas para as réplicas por multicast, sendo a resposta devolvida ao cliente quando for recebido o número de respostas configurado. A replicação passiva foi aplicada aos serviços Web por uma infraestrutura com arquitetura modular [21], facilmente extensível para replicação ativa ou *coordinator-cohoort*. O Perpetual-WS [22] baseia-se no protocolo CLBFT [6] e suporta operações de longa duração, bem como operações não-determinísticas, uma vez que o serviço replicado chegará a consenso relativamente à resposta a devolver aos clientes. Um algoritmo BFT [16] que apenas necessita de ordenação de mensagens parcial, ou seja, por emissor, obtendo melhor desempenho que outros protocolos, foi proposto em [7] para alcançar coordenação fiável em WS-BusinessActivity, limitando-se a estas atividades. O modelo de N Versões é aplicado por WS-FTM [17] a um serviço Web para aumentar a sua confiabilidade, permitindo-lhe tolerar faltas físicas e bizantinas.

6 Conclusão e Trabalho Futuro

A coordenação de serviços com tolerância a faltas assenta na disseminação de informação de forma confiável. A disseminação de informação no contexto das arquiteturas orientadas a serviços envolvendo muitos dispositivos apresenta um conjunto de desafios que não são adequadamente ultrapassados com abordagens tradicionais. Para ultrapassá-los, propomos a utilização de gossip ao nível arquitetural, que apresenta várias vantagens, como a utilização de diferentes variantes com baixa complexidade, ao mesmo tempo que fornece fortes garantias de disseminação atómica e confiável. Contrastando com abordagens anteriores [25], a

nossa proposta integra-se perfeitamente num ambiente Devices Profile for Web Services (DPWS), sendo compatível com os dispositivos existentes.

Outra abordagem para lidar com faltas de servidores e de comunicações é a utilização de um protocolo de consenso. Assim, implementando o protocolo Raft sobre DPWS, obtivemos um serviço de coordenação semelhante ao Apache ZooKeeper que permite a replicação de serviços.

Este artigo descreve uma infraestrutura implementada sobre WS4D JMEDS que inclui os serviços WS-Gossip e Raft4WS. Isto permitiu demonstrar que o desempenho de gossip sobre SOAP-over-UDP, incluído em DPWS, num cenário de um para muitos, é superior ao de WS-ReliableMessaging e WS-Eventing tanto em termos de latência como de tolerância a faltas, oferecendo flexibilidade e resiliência adicionais. Para além disto, o WS-Gossip possibilita a disseminação de muitos para muitos, bem como a recolha agregada de informação de muitos para um, através das diversas operações disponibilizadas.

Como trabalho futuro, pretendemos proceder à avaliação das restantes variantes e do efeito de *churn* e de múltiplas fontes de eventos na rede, bem como da extrapolação dos parâmetros tendo em conta o cenário.

Agradecimentos

Este trabalho foi parcialmente financiado pelo Projeto Smartgrids - NORTE-07-0124-FEDER-000056 cofinanciado pelo Programa Operacional Regional do Norte (ON.2 – O Novo Norte), ao abrigo do Quadro de Referência Estratégico Nacional (QREN), através do Fundo Europeu de Desenvolvimento Regional (FEDER).

Referências

1. Minha: Middleware Testing Platform. `http://www.minha.pt/`.
2. SOAP over Java Message Service 1.0 W3C Recommendation. `http://www.w3.org/TR/2012/REC-soapjms-20120216/`, 16 February 2012.
3. K. Birman. A review of experiences with reliable multicast. *Software Practice and Experience*, 29(9), July 1999.
4. F. Campos and J. Pereira. Gossip-based service coordination for scalability and resilience. In *Proc. 3rd Workshop on Middleware for service oriented computing*, MW4SOC '08, 2008.
5. N. A. Carvalho, J. a. Bordalo, F. Campos, and J. Pereira. Experimental evaluation of distributed middleware with a virtualized java environment. In *Proc. 6th Workshop on Middleware for Service Oriented Computing*, MW4SOC '11, 2011.
6. M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proc. 3rd Symp. on Operating Systems Design and Implementation*, pages 173–186, 1999.
7. H. Chai, H. Zhang, W. Zhao, P. Melliar-Smith, and L. Moser. Toward trustworthy coordination of web services business activities. *Services Computing, IEEE Trans.*, 6(2):276–288, 2013.
8. Q. Chen, H. Ghenniwa, and W. Shen. Web-services infrastructure for information integration in power systems. In *Power Engineering Society General Meeting*, 2006.

9. K. De Brabandere, K. Vanthournout, J. Driesen, G. Deconinck, and R. Belmans. Control of microgrids. In *Power Engineering Society General Meeting*, 2007.
10. P. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulie. Epidemic information dissemination in distributed systems. *Computer*, 37(5):60–67, May 2004.
11. D. Ferguson, T. Storey, B. Lovering, and J. Shewchuk. Secure, Reliable, Transacted Web Services. `http://www.ibm.com/developerworks/webservices/library/ws-securtrans/index.html`, 2003.
12. F. Forno and P. Saint-Andre. SOAP Over XMPP 1.0 specification. `http://xmpp.org/extensions/xep-0072.html`, 14 December 2005.
13. P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. *Proc. 2010 USENIX Annual Tech. Conf.*, 8, 2010.
14. S. Karnouskos. Asset monitoring in the service-oriented internet of things empowered smartgrid. *Service Oriented Computing and Applications*, 6(3), 2012.
15. A. Krkoleva, V. Borozan, A. Dimeas, and N. Hatziargyriou. Requirements for implementing gossip based schemes for information dissemination in future power systems. In *Innov. Smart Grid Tech., 2nd IEEE PES Int. Conf. and Exh. on*, pages 1–7, 2011.
16. L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Programming Languages and Systems*, 4(3):382–401, 1982.
17. N. Looker, M. Munro, and J. Xu. Increasing web service dependability through consensus voting. *Proc. 29th Int. Conf. Computer Software and Applications*, 2005.
18. N. Mohamed and J. Al-Jaroodi. A survey on service-oriented middleware for wireless sensor networks. *Service Oriented Computing and Applications*, 5(2), 2011.
19. E. Newcomer and G. Lomow. *Understanding SOA with Web Services.* Addison Wesley Professional, 2004.
20. D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. *ramcloud.stanford.edu*, 2013.
21. J. Osrael, L. Froihofer, M. Weghofer, and K. Goeschka. Axis2-based replication middleware for web services. *Web Services. IEEE Int. Conf.*, pages 591–598, 2007.
22. S. Pallemulle, H. Thorvaldsson, and K. Goldman. Byzantine fault-tolerant web services for n-tier and service oriented architectures. In *Distributed Computing Systems. 28th Int. Conf. on*, pages 260–268, 2008.
23. J. Pereira and R. Oliveira. The mutable consensus protocol. *Reliable Distributed Systems. Proc. 23rd IEEE Int. Symp. on*, pages 218–227, 2004.
24. R. Piantoni and C. Stancescu. Implementing the swiss exchange trading system. *27th Int. Symp. Fault-Tolerant Computing*, pages 309–313, 1997.
25. R. V. Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.
26. J. Salas, F. Perez-Sorrosal, M. Patiño-Martínez, and R. Jiménez-Peris. WS-Replication: A Framework for Highly Available Web Services. In *15th Intl. World Wide Web Conf. (WWW. 06)*, 2006.
27. I. Stoica, R. Morris, D. Karger, and M. Kaashoek. Chord: A scalable peer-to-peer lookup service for internet applications. *Proc. 2001 Conf. on Applications*, 2001.
28. W. Vogels and C. Re. WS-Membership - Failure Management in a Web-Services World. *Intl. World Wide Web Conf. (WWW. 2003)*, 2003.