Yggdrasil: Uma Framework para Desenvolvimento e Execução de Protocolos em Redes Ad Hoc*

Pedro Ákos Costa and João Leitão

NOVA LINCS & DI/FCT/NOVA University of Lisbon, Lisboa, Portugal

Resumo As redes ad hoc sem fios foram extensivamente estudadas no passado, levando ao aparecimento de protocolos distribuídos para resolver vários desafios como o encaminhamento, agregação e disseminação de dados, entre outros. A maioria destes protocolos e soluções foram validadas recorrendo a simulações. No entanto, o aparecimento de novas aplicações, particularmente no domínio da Internet das Coisas (*Internet of Things* (IoT)) e de computação na berma (*edge computing*), leva à necessidade de revisitar estes problemas e soluções, e desenvolver soluções práticas que possam executar em dispositivos comuns. Infelizmente, existe uma significativa falta de ferramentas de apoio ao desenvolvimento deste tipo de protocolos que permitam não só a sua rápida prototipagem mas também, a sua execução eficiente para avaliação ou para o suporte da execução de aplicações.
Para colmatar esta lacuna, neste artigo propomos uma nova *framework* que pretende simplificar o desenvolvimento e execução de protocolos e aplicações distribuídas para redes ad hoc sem fios em dispositivos comuns. A nossa *framework*, denominada Yggdrasil, proporciona aos programadores um ambiente de desenvolvimento simples e efetivo. Isto é alcançado através de um conjunto de abstrações que incluem: um modelo de execução baseado em eventos, primitivas de comunicação ao nível físico, mecanismos de interação entre protocolos/aplicações e mecanismos para execução *multi-threaded*. Um protótipo totalmente funcional foi desenvolvido na linguagem C e testado experimentalmente num conjunto de 24 Raspberry Pis.

Keywords: Redes Ad Hoc · Protocolos Distribuídos · Frameworks.

1 Introdução

As redes sem fios ad hoc foram extensivamente estudadas no passado devido ao seu potencial de escalabilidade e de facilidade de instalação, e da sua adequação a cenários onde o uso de infra-estrutura é impossível, como cenários de guerra [17], catástrofes naturais [3], ou outros. Neste contexto, múltiplos protocolos foram propostos pela comunidade científica, desde protocolos de encaminhamento [4],

* Trabalho parcialmente financiado pelo projeto europeu LightKone (H2020 grant agreement ID 732505) e NOVA LINCS (FC&T UID/CEC/04516/2013)

e de difusão de informação tolerante a falhas [10], e até mesmo de acordo bizantino [12]. No entanto o número de aplicações práticas baseadas em redes ad hoc, nomeadamente aplicações que envolvam grandes números de dispositivos a interagir e colaborar activamente, não é muito significativo.

Porém, o recente aumento de popularidade e interesse em aplicações no domínio da Internet das Coisas (*Internet of Things* ou simplesmente *IoT*) motiva a necessidade de revisitar estes problemas. Particularmente devido ao facto de a grande maioria destas aplicação atualmente recorrer a soluções baseadas na nuvem, cuja capacidade de receber, processar, e emitir respostas em tempo útil para a crescente quantidade de dados gerada por estas aplicações, está a alcançar o seu limite. Nomeadamente devido à limitada largura de banda entre os dispositivos clientes (que produzem e consomem dados) e as infra-estruturas de nuvem. Considerando que se espera um aumento significativo do volume de dados produzido por estas aplicações no futuro próximo [2], torna-se essencial descentralizar o processamento de dados de aplicações IoT.

O modelo de computação na berma (*edge computing*) [15] apresenta-se como uma alternativa viável, ao promover o deslocamento de computações e armazenamento destas aplicações para fora dos limites dos centros de dados, executando estas tarefas mais próximo dos clientes finais, potencialmente (mas não exclusivamente) nos dispositivos dos próprios clientes. Neste sentido, e em alguns cenários particulares, nomeadamente em cenários aplicacionais relacionados com cidades inteligentes, espaços inteligentes, e até mesmo agricultura e extração de recursos naturais inteligente, poderá ser conveniente suportar a interação directa entre inúmeros dispositivos na berma sem acesso a infra-estruturas de rede. Neste caso, o uso de redes ad hoc e vários protocolos distribuídos para este cenário torna-se uma alternativa não só viável mas também interessante. Adicionalmente, o desenvolvimento de novos protocolos e soluções para este cenário, poderá potenciar o aparecimento de novas aplicações com componentes na berma, nomeadamente aplicações móveis e altamente interactivas [16,5].

Infelizmente, existe uma significativa falta de ferramentas de suporte ao desenvolvimento e execução de protocolos e aplicações que alavancam em redes ad hoc sem fios, nomeadamente para dispositivos de uso comum (i.e, microcomputadores a executarem sistemas operativos comuns). Adicionalmente, muitos dos protocolos e soluções desenvolvidos para estes ambientes foram maioritariamente testados através de ferramentas de simulação [14], sendo potencialmente necessária uma melhor compreensão do seu funcionamento em ambientes reais (por exemplo ambientes altamente poluídos por sinais de rádio).

De forma a endereçar estas limitações, neste artigo apresentamos o Yggdrasil, uma nova *framework* para suportar o desenvolvimento, avaliação, e execução de protocolos e aplicações em redes ad hoc. Esta *framework* proporciona aos programadores um ambiente de desenvolvimento simples e efetivo. Isto é alcançado através de um conjunto de abstrações que incluem: um modelo de execução baseado em eventos, primitivas de comunicação ao nível físico, mecanismos de interação entre protocolos/aplicações e mecanismos para execução *multi-threaded*. Um protótipo da *framework* foi desenvolvida em C, de forma a permitir a execução de protocolos e aplicações não só em dispositivos de uso

comum (e.g, portáteis, micro-computadores) mas também para potenciar a sua execução em dispositivos com recursos mais limitados como sensores.

Neste artigo, utilizamos um protocolo de encaminhamento em redes ad hoc sem fios como motivação para os requisitos da *framework*, de forma a guiar o seu desenvolvimento, e também como demonstrador do esforço relativo na implementação de um protocolo de média complexidade e da eficácia da *framework* no suporte da execução prática deste protocolo e de uma aplicação de teste simples.

O resto deste artigo, está organizado da seguinte forma: a secção 2 apresenta um levantamento de requisitos para o desenho do Yggdrasil baseado no caso de estudo das necessidades do protocolo de encaminhamento B.A.T.M.A.N. [4]; a secção 3 descreve conceptualmente a *framework* Yggdrasil; a secção 4 detalha a implementação de um protocolo exemplo no Yggdrasil; a secção 5 apresenta os detalhes de trabalho experimental em que recorremos a um conjunto de 24 Raspberry Pi 3 - Modelo B; a secção 6 discute brevemente trabalho relacionado; e por fim, a secção 7 concluí o artigo.

2 Requisitos

Consideramos o protocolo de encaminhamento para redes ad hoc sem fios *Better Approach To Mobile Ad hoc Networking* , ou simplesmente B.A.T.M.A.N. [4], que constrói uma tabela de encaminhamento para refletir a ligação mais estável para encaminhar uma mensagem para um destino na rede. Para conseguir identificar as ligações mais estáveis, no protocolo B.A.T.M.A.N. cada nó envia periodicamente anúncios contendo um número de sequência, incrementado pelo emissor a cada emissão. Estes anúncios são difundidos pela rede, sendo marcados nos recetores a identificação do nó que originou o anúncio, o número de sequência do anúncio, e o nó que gerou a retransmissão desse anúncio recebida pelo nó local. Os números de sequência recebidos são mantidos numa janela deslizante, sendo que a ligação mais estável para um destino (i.e, o próximo salto (*hop*) para cada destino possível) é definido como sendo o vizinho do nó do qual foi recebido anúncios contendo mais números de sequência distintos dentro da janela. A janela de receção de cada nó avança independentemente para cada destino distinto sempre que um número de sequência mais alto do que o limite atual da janela é recebido.

A própria natureza de um protocolo de encaminhamento aponta numa direção relevante. Várias aplicações, ou até outros protocolos, para redes ad hoc podem necessitar do uso de um protocolo de encaminhamento. Esta observação carrega a consequência (natural) de que a *framework* deve suportar a definição e execução de protocolos sob a forma de *módulos auto-contidos*, por forma a permitir a sua reutilização por várias aplicações concretas.

Qualquer protocolo distribuído, sendo que o B.A.T.M.A.N. não é uma exceção, necessita de abstrações de comunicação entre os vários nós. Em particular, no contexto deste protocolo, a difusão de anúncios para todos os vizinhos de um nó requer o suporte a primitivas de comunicação que alavanquem no uso de *difusão a um-salto* (em inglês *one hop broadcast*) [18]. No entanto, e considerando o encaminhamento específico de mensagens aplicacionais para um destino concreto, este é tratado pelo B.A.T.M.A.N. em cada nó através do re-envio da mensagem para o vizinho considerado como o próximo melhor salto, até à mensagem

chegar ao seu destino. Assim torna-se também relevante fornecer *primitivas de comunicação directa (i.e, unicast) para nós ao alcance do rádio (i.e, vizinhos).*

Deve-se notar que as mensagens encaminhadas pelo protocolo B.A.T.M.A.N. até um destino concreto não são geradas pelo próprio protocolo. Tipicamente, espera-se que estas mensagens sejam geradas por outros protocolos ou aplicações que delegam no protocolo de encaminhamento a responsabilidade de fazer chegar a mensagem ao destinatário correcto. Isto implica que é necessário a existência de um mecanismo que permita a interação entre protocolos ou aplicações que coexistem num único nó (dentro da *framework* Yggdrasil). Desta forma suportamos um mecanismo de interação baseados em *pedidos* que requerem uma ação de um protocolo concreto (por exemplo o envio e encaminhamento de uma mensagem). Alguns destes pedidos podem requerer uma resposta, pelo que suportamos a opção do envio de uma *resposta* de volta ao protocolo ou aplicação que gerou o pedido (por exemplo, o protocolo B.A.T.M.A.N. poderia notificar que o destino para o qual lhe é solicitado o encaminhamento de uma mensagem é desconhecido).

Uma consequência, talvez menos óbvia, da operação do protocolo B.A.T.M.A.N. na criação da sua tabela de encaminhamento local é que o protocolo fica na posse de informação relativa à filiação da rede, efetivamente obtendo a mesma informação que alguns protocolos de descoberta de nós. Notamos que em alguns contextos, aplicações ou outros protocolos no nó local poderiam beneficiar desta informação, no entanto durante a implementação do protocolo é impossível saber que protocolos/aplicações em tempo de execução podem necessitar dessa informação. Isto motiva-nos a suportar um segundo mecanismo de interação entre protocolos/aplicações no mesmo nó, onde é possível *emitir uma notificação de um evento* (e.g, a descoberta de um novo nó no sistema) que é automaticamente entregue pela *framework* a cada protocolo ou aplicação que tenha anteriormente registado o seu interesse em receber esse tipo de notificação.

Um outro comportamento exibido pelo B.A.T.M.A.N. que é relevante é a transmissão periódica de mensagens. Este comportamento é frequente noutros protocolos, sendo que alguns podem inclusive necessitar de tomar ações ao final de um certo intervalo de tempo, por exemplo para lidar com falhas de outros nós. Esta observação motiva-nos a fornecer um mecanismo de gestão de temporizadores ao nível da *framework*. De forma a capturar os usos frequentes de temporizadores em protocolos distribuídos, este mecanismo deve suportar o registo de *temporizadores periódicos* que são emitidos para um protocolo ou aplicação recorrentemente com um intervalo temporal constante e *temporizadores únicos* que são emitidos para um protocolo ou aplicação uma única vez após um tempo parametrizável desde o momento da sua criação.

Finalmente, uma outra observação que podemos fazer, apesar de não estritamente relacionada com a operação do protocolo B.A.T.M.A.N., é que muitas vezes existem vários protocolos que podem oferecer o mesmo serviço a outros protocolos/aplicações. Frequentemente, estas diferentes alternativas têm comportamentos ou desempenhos diferentes de acordo com aspectos específicos do seu ambiente de execução (que apenas podem ser inferidos em tempo de execução). Isto motiva-nos a suportar na *framework* um mecanismo que permita em tempo

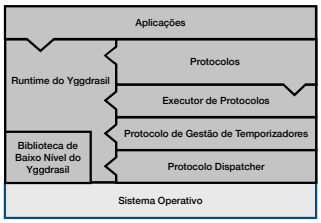


Figura 1. Visão simplificada da arquitectura do Yggdrasil.

de execução *activar e desactivar protocolos/aplicações*, como forma de pavimentar o caminho à construção de soluções com mecanismos de gestão autonómica. Outro aspeto relevante prende-se com o desempenho geral de aplicações compostas por vários protocolos que competem por recursos do sistema na sua execução, nomeadamente tempo de CPU. De forma a poder alavancar em dispositivos com vários CPUs, suportamos a hipótese de protocolos/aplicações executarem num modo em que têm um fio de execução (*thread*) próprio ou em que vários protocolos partilham um único fio de execução. De acordo com as preferências do programador a *framework* é responsável por, transparentemente, *gerir a execução dos vários fios de execução* escondendo problemas de concorrência que poderiam ocorrer (e.g, vários protocolos que tentam enviar uma mensagem pelo rádio ao mesmo tempo).

3 Yggdrasil

A figura 1 apresenta esquematicamente a arquitectura da *framework*. Central ao desenho do Yggdrasil é a noção de protocolo e aplicação. Do ponto de vista da *framework*, ambos são tratados da mesma forma. Na nossa *framework*, e de uma forma muito simplificada, cada protocolo/aplicação é visto como uma máquina de estados, que contém um estado interno, apenas manipulado por si mesmo, e que consome eventos. Estes eventos produzem modificações no estado interno do protocolo/aplicação ou expõem porções desse estado interno. Os eventos que guiam a execução dos protocolos são de quatro tipos: *i*) **Mensagens**, que são o único tipo de evento que pode ser transportado entre protocolos/aplicações que executem em instâncias independentes do Yggdrasil (tipicamente em nós diferente); *ii*) **Temporizadores**, que notificam um protocolo da necessidade de executar uma tarefa periódica ou da expiração de um timeout local; *iii*) **Pedidos e Resposta**, que permitem a interação directa entre protocolos/aplicações no contexto de um único processo Yggdrasil; e finalmente *iv*) **Notificações**, que permitem a interação indirecta entre protocolos/aplicações no contexto de um único processo Yggdrasil.

Cada protocolo/aplicação possuí, internamente uma fila de eventos onde os eventos gerados/recebidos para esse protocolo são guardados. A fila é gerida pelo Yggdrasil e os eventos são consumidos individualmente por cada protocolo ou aplicação. Nem todos os eventos têm a mesma prioridade no consumo. Isto quer dizer que quando eventos de tipos diferentes se encontram na fila de eventos de um protocolo, estes são consumidos de acordo com o seu tipo, pela ordem: temporizadores, notificações, mensagens, pedidos/respostas. A lógica para esta ordenação prende-se com o prioritizar eventos que podem ter maior sensibilidade

a atrasos de processamento. Evidentemente um protocolo/aplicação pode emitir eventos (para si ou para outros componentes) enquanto processa outros eventos. Todos os protocolos devem também implementar um mecanismo de inicialização, onde inicializam o seu estado interno e potencialmente geram um conjunto de eventos iniciais (e.g, gerar um temporizador periódico para despoletar uma ação periódica da sua parte).

A *framework* oferece tipos de dados específicos para representar cada um dos eventos referidos acima. Na sua maioria, estes tipos de dados incluem uma zona de *payload* (que é gerida pelo protocolo ou aplicação que cria/consome o evento) e uma zona com cabeçalhos que codificam informação relativa aos eventos usada pela *framework* para processar esses eventos, como por exemplo o protocolo de destino, ou identificadores (únicos) internos.

No modelo de execução por defeito do Yggdrasil cada protocolo/aplicação é executado no contexto de um fio de execução próprio, isto permite alavancar na existência de múltiplos CPUs em dispositivos, e permite que protocolos/aplicações diferentes evoluam simultaneamente. No entanto, o programador não tem, na generalidade, de se preocupar com problemas devido a concorrência, visto a *framework* promover um modelo de desenvolvimento independente dos protocolos e de interações através das filas de eventos que são geridas internamente pela *framework*.

Para lidar com casos em que os dispositivos tem poucos CPUs ou para permitir uma gestão dinâmica em tempo de execução dos protocolos que se encontram activos num processo, oferecemos também um executor de protocolos. Este executor é um *pseudo-protocolo* que internamente partilha um fio de execução com múltiplos protocolos. De uma forma simplificada, o executor mantém informação sobre as filas de eventos dos protocolos que gere, e processa eventos de cada protocolo (utilizando a lógica do protocolo especifico) à vez.

Finalmente existem dois protocolos internos ao Yggdrasil e que tipicamente farão parte de qualquer aplicação desenvolvida com base na *framework*. Estes protocolos são o *Dispatcher* e o *Gestor de Temporizadores*. O Dispatcher é responsável por gerir o envio e receção de mensagens para a rede. Visto o Yggdrasil ter sido especificamente desenvolvido para redes ad hoc, o Dispatcher recorre a uma biblioteca de baixo nível do Yggdrasil para aceder directamente ao rádio do dispositivo. Esta biblioteca permite configurar o rádio do dispositivo para operar em modo ad hoc, fornecendo uma abstração de um canal. Este canal é configurado para efectuar o envio e receção de mensagens no nível físico, sendo assim possível suportar primitivas de comunicação de difusão em um salto e ponto-a-ponto. No entanto, para este fim o canal interage com a interface de rede em modo promiscuo. Para evitar receber mensagens que não são enviadas por algum protocolo/aplicação Yggdrasil, o canal é também configurado para ter um filtro ao nível do sistema operativo que apenas receciona as mensagens que contêm um cabeçalho definido no filtro.

Tipicamente as mensagens são enviadas entre protocolos idênticos em processos diferentes. É responsabilidade do Dispatcher serializar mensagens para a rede, receber e desserializar as mensagens da rede, entregando-as ao protocolo adequado. Adicionalmente, e apesar de não discutirmos esse aspeto em detalhe,

a *framework* oferece também suporte à interceção de filas de protocolos. Este mecanismo por exemplo, permite a um protocolo transparentemente capturar mensagens que são enviadas para a rede (e consequentemente direcionadas ao protocolo Dispatcher) de forma a adicionar informação extra (e.g, piggybacking).

O Yggdrasil oferece ainda a hipótese ao programador de uma aplicação, de substituir o protocolo de Dispatcher por uma variante sua. Esta funcionalidade é particularmente útil para efectuar testes ou depuramento de protocolos/aplicações, através do enriquecimento deste protocolo com mecanismos de log ou de validação semântica de mensagens.

O protocolo de gestão de temporizadores, permite a vários protocolos/aplicações num processo Yggdrasil registar pedidos de notificação periódicos ou únicos. Para além disso, e com base em identificadores únicos associados a cada temporizador, estes pedidos podem ser também cancelados através do envio de um pedido a este protocolo. De forma a manter a interface de interação entre os vários protocolos e aplicações com o Gestor de Temporizadores simples, todos os pedidos feitos ao gestor de temporizador são feitos através do envio de estruturas de dados do tipo temporizador.

Finalmente, notamos que o runtime do Yggdrasil oferece uma API com o objectivo de simplificar o desenho e implementação de protocolos e aplicações. Esta API inclui, entre outros, mecanismos para registar/cancelar temporizadores, emitir eventos (pedidos, respostas, notificações, e mensagens) entre outros.

4 Desenvolvimento no Yggdrasil

Foi desenvolvido um protótipo do Yggdrasil na linguagem C. Este protótipo inclui todas as funcionalidades descritas anteriormente e implementações de vários protocolos de suporte adicionais. Como, protocolos de descoberta de vizinhos, deteção de falhas, entre outros. O protótipo inclui ainda bibliotecas com funções auxiliares para a criação e manipulação dos vários tipos de eventos. No total, o protótipo atual contém mais de 20.000 linhas de código.

O runtime no protótipo atual é visto como uma biblioteca usada na implementação de protocolos e aplicações. Um processo Yggdrasil é composto por uma aplicação principal que define uma configuração de rede (modo de operação do rádio, frequência, nome da rede, etc). Esta configuração é utilizada na inicialização do runtime através do uso da biblioteca de baixo nível fornecida. Adicionalmente o runtime inicializa ainda os protocolos de suporte do Yggdrasil previamente mencionados (*Dispatcher*, *Gestor de Temporizadores* e o *Executor de Protocolos*).

A figura 2 apresenta o código C de um protocolo exemplo. Este protocolo demonstra como utilizar as várias abstrações fornecidas pela *framework*. De uma forma simplificada, o protocolo opera da seguinte forma, periodicamente (ao receber um temporizador) um nó envia uma mensagem utilizando difusão a um salto (*one hop broadcast*) contendo uma string pré-definida (fig. 2: 18-24). Quando um nó recebe uma destas mensagens, é enviado uma notificação (com o conteúdo da mensagem) que é entregue (através da *framework*) a todos os protocolos locais que tenham indicado interesse nesse tipo de notificação (fig. 2: 25-31). O protocolo processa ainda uma notificação (gerada por um protocolo de descoberta de vizinhos) onde regista informação sobre o vizinho, cuja informação é

```
1  short process_request(YggRequest* request) {
2    YggRequest reply;
3    YggRequest_init(&reply, PROTO_MY_PROTO, request->proto_origin, REPLY, REPORT);
4    YggRequest_addPayload(&reply, &msg_count, sizeof(int));
5    deliverReply(&reply);
6    YggRequest_freePayload(request);
7    YggRequest_freePayload(&reply);
8  }
9  short process_event(YggEvent* event) {
10   uuid_t neighbour_id;
11   WLANAddr neighbour_addr;
12   void* ptr = YggEvent_readPayload(event, NULL, neighbour_id, sizeof(uuid_t));
13   YggEvent_readPayload(event, ptr, neighbour_addr.data, WLAN_ADDR_LEN);
14   neighbour_list* newnei = new_neighbour(neighbour_id, neighbour_addr, NULL, 0);
15   neighbour_add_to_list(&neighbours, newnei);
16   YggEvent_freePayload(event);
17 }
18 short process_timer(YggTimer* timer) {
19   char* hello = "Hello from the other side";
20   YggMessage msg;
21   YggMessage_initBcast(&msg, PROTO_MY_PROTO);
22   YggMessage_addPayload(&msg, hello, strlen(hello)+1);
23   dispatch(&msg);
24 }
25 short process_msg(YggMessage* msg) {
26   YggEvent ev;
27   YggEvent_init(&ev, PROTO_MY_PROTO, MSG);
28   YggEvent_addPayload(&ev, msg->data, msg->datalen);
29   deliverEvent(&ev);
30   YggEvent_freePayload(&ev);
31 }
32 void main_loop(queue_t* inBox) {
33   while(1) {
34       queue_t_elem elem;
35       queue_pop(inBox, &elem);
36       switch(elem.type) {
37         ...
38       }
39       free_elem_payload(&elem);
40   }
41 }
42 proto_def* my_proto_init(void* args) {
43   proto_def* proto =  create_protocol_definition(PROTO_MY_PROTO, "MyProto", &
       my_proto_destroy);
44   if(run_in_executor_mode(args)) {
45     proto_def_add_msg_handler(proto, &process_msg);
46     proto_def_add_timer_handler(proto, &process_timer);
47     proto_def_add_event_handler(proto, &process_event);
48     proto_def_add_request_handler(proto, &process_request);
49   } else {
50     proto_def_add_protocol_main_loop(proto, &main_loop);
51   }
52   proto_def_produced_events(proto, 1);
53   proto_def_add_consumed_event(proto, PROTO_DISCOV, NEW_NEIGH);
54   YggTimer timer;
55   YggTimer_init(&timer, 2, 500000, 5, 0);
56   setupTimer(&timer);
57   return proto;
58 }
```

Figura 2. Exemplo da implementação de um Protocolo (brinquedo).

providenciada na notificação (fig. 2: 9-17). Finalmente, este protocolo ainda suporta pedidos (enviados por outros protocolos ou aplicações locais). Este pedidos permitem ao protocolo responder com o número de mensagens enviadas até ao momento (fig. 2: 1-8).

De forma a ilustrar as duas formas de operação descritas anteriormente, apresentamos ainda uma versão (parcial) da implementação do ciclo de controle principal (fig.. 2: 32-41), a qual faz o protocolo esperar continuamente por um evento na sua fila de eventos, analisar o tipo de evento e invocar a função de processamento desse tipo de evento (descritas acima).

Finalmente, ilustramos ainda a função de inicialização protocolo que fornece ao runtime uma estrutura de dados com todos os dados necessários à sua execução (fig.. 2: 42-58). Esta estrutura de dados pode ser preenchida de duas formas, de acordo com uma opção indicada nos argumentos desta função. No caso

do protocolo ir executar no contexto do *Executor de Protocolos*, as funções que processam cada tipo de evento são registadas, caso contrário, apenas a função de controle principal é registada. Esta função de inicialização é ainda responsável por ações adicionais, como o registo do temporizador periódico, que dispara pela primeira vez após $2.5s$, e depois a cada $5s$.

Apesar deste protocolo ser artificial, no sentido de não fornecer um serviço útil, a sua implementação demonstra a flexibilidade da *framework* a suportar vários comportamentos distintos. No total este protocolo (incluíndo as partes omitidas no fig. 2) tem cerca de 100 linhas.

5 Avaliação Experimental

Nesta secção reportamos a avaliação experimental da nossa *framework*. Esta avaliação está dividida em dois pontos. No primeiro discutimos a nossa implementação do protocolo de encaminhamento que serve de caso de estudo neste artigo (B.A.T.M.A.N.), comparando-a informalmente com uma implementação real do mesmo protocolo em C (que opera como um daemon). De seguida conduzimos uma avaliação de desempenho deste protocolo usando um conjunto de 24 Raspberry Pi 3 - Modelo B, que demonstra a exequibilidade do uso do Yggdrasil para executar protocolos e aplicações em ambientes reais.

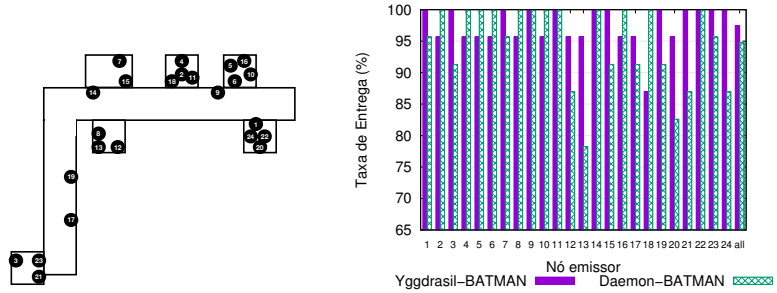
5.1 Implementação do B.A.T.M.A.N.

De forma a validar a utilidade do Yggdrasil na implementação de protocolos que operam em redes ad hoc, fizemos uma implementação da versão 4 do protocolo B.A.T.M.A.N. [13]. Esta implementação faz um conjunto de simplificações simples, mas que não afetam a execução ou correção do algoritmo. Em particular, a implementação das janelas deslizantes do protocolo recorre a uma representação de dados expandida (ao invés de mascaras de bits utilizadas na implementação de referência) e o tamanho das mensagens trocadas entre processos é ligeiramente maior do que o tamanho das mensagens originalmente especificadas.

A nossa implementação, incluindo testes preliminares e correções à implementação original, demorou cerca de três dias. Interessantemente, os erros existentes estavam relacionados com a gestão das janelas deslizantes utilizadas para registar os anúncios recebidos, e não nas componentes de interação suportadas pelo Yggdrasil directamente. A nossa implementação tem menos de 700 linhas de código C, incluindo algum código usado na depuração de erros.

Um aspecto interessante desta implementação é a possibilidade de compara-la com uma implementação de referência, também na linguagem C, que opera como um daemon linux[1]. O daemon implementa a versão 3 do protocolo B.A.T.M.A.N. que, ignorando aspectos relativos ao suporte de acesso à Internet e redes externas (também ignorados pela nossa implementação), possuí a mesma lógica que a versão 4. Analisando o código desta implementação com algum detalhe, descobrimos que possuí aproximadamente 7.000 linhas de código C. Retirando a parte dedicada à interação com o kernel, suporte de acesso à Internet e redes externas, e outras otimizações ignoradas pela nossa implementação, estimamos que aproximadamente 2.000 linhas são dedicadas à lógica principal de encaminhamento. Isto representa um fator de aumento de quase 3 vezes no número de linhas de

[1] Disponível em https://www.open-mesh.org/projects/open-mesh/wiki



(a) Disposição dos Dispositivos (b) Taxa de Entrega de Mensagens

Figura 3. Configuração Experimental

código considerando a nossa implementação. Esta diferença é maioritariamente devida ao facto do daemon ter de lidar com aspetos de baixo nível, como por exemplo a serialização de mensagens, gestão de temporizadores, gestão de concorrência, gestão de interfaces de rede, entre outros, que são simplificados pelo uso da nossa *framework*.

Estas observações permitem-nos concluir que de facto, a *framework* Yggdrasil oferece abstrações úteis ao desenvolvimento de protocolos e aplicações distribuídas para redes ad hoc. Outra observação interessante, é que esta simplicidade poderá permitir o uso da *framework* na lecionação de disciplinas avançadas nas quais se discutam redes ad hoc e protocolos distribuídos.

5.2 Avaliação de Desempenho

A nossa avaliação experimental recorre a uma simples aplicação de demonstração que recorre à nossa implementação do B.A.T.M.A.N. e ao daemon do B.A.T.M.A.N. para, periodicamente, enviar mensagens para nós na rede selecionados de forma aleatória. Quando um nó recebe uma destas mensagens, envia para o emissor uma mensagem de confirmação da receção da mesma. A nossa aplicação regista um histórico de todas as mensagens enviadas e recebidas que processámos off-line depois de executar as experiências.

As experiências foram efetuadas num conjunto de 24 Raspberry Pi 3 - Modelo B espalhados pelo nosso departamento em salas ao longo de dois corredores, cada um com cerca de $30m$. A figura 3a apresenta um esquema da distribuição dos dispositivos. A aplicação foi configurada para enviar uma mensagem com 50% de probabilidade a cada $2s$ de execução. Foi executado um processo com a configuração descrita anteriormente em cada dispositivo. Cada experiência foi executada por um período de dez minutos.

A figura 3b apresentada a taxa de entrega em todos os nós do sistema, tanto para a nossa implementação do B.A.T.M.A.N., apontado na figura como *Yggdrasil-BATMAN*, como para o daemon do B.A.T.M.A.N., e um último par de colunas adicional com a média da taxa de entrega de todos os nós (com a etiqueta *all* (todos)). Notamos que em cada experiência aproximadamente 3.500 mensagem foram enviadas, sendo que cada nó enviou à volta de 150. Os resultados mostram que as duas implementações são equivalentes, sendo que as taxas de entrega média diferem em apenas 3% a favor da nossa implementação. Esta diferença poderá se dever à interferência no meio sem fios. Alguns nós (como

por exemplo o nó 20) têm sistematicamente taxas de entrega inferiores (i.e, as mensagens enviadas por eles chegam ao seu destino com menor frequência). Isto é devido ao facto de alguns nós estarem posicionados perto de pontos de acesso WiFi ou perto de portáteis com rádios WiFi ativos, que frequentemente ocupam o meio sem fios.

6 Trabalho Relacionado

Outras *frameworks* e *toolkits* existem mas não específicos para as redes ad hoc sem fios. Grande parte das *frameworks* existentes foram desenhadas para suportar comunicação em grupo, ou para fornecer propriedades de deteção de falhas, no entanto estas, apesar de assumir um modelo semelhante ao nosso, não consideram os desafios adicionais e oportunidades oferecidas pelos canais sem fios. Outras *frameworks* que se especializam em canais sem fios, foram construídas para suportar o desenvolvimento e execução de aplicações em hardware especializado como por exemplo sensores.

O Appia [11] é uma *framework* desenvolvida para suportar comunicação em grupo, garantindo a qualidade de serviço das aplicações nele construídas. O Appia apresenta um modelo de execução de protocolos semelhante ao nosso, sendo baseado em eventos, mas não tendo o suporte para múltiplos fios de execução. Todos os protocolos usados numa aplicação do APPIA são executados no contexto de um único fio de execução, o que tornar a execução pouco eficiente quando parte dos protocolos executam computações complexas.

O TinyOs [6] é um sistema operativo leve para desenvolver aplicações para redes de sensores, como tal, está totalmente preparado para canais sem fios. As aplicações e protocolos são desenvolvidos também num modelo de execução baseado em eventos, sendo fornecido bastante suporte pelo TinyOs. No entanto, o TinyOs foi desenvolvido a pensar num modelo de execução único.

O Impala [7] é a *framework* que mais se assemelha ao Yggdrasil, sendo um *middleware* que age como um sistema operativo leve, mas cujo focos é na atualização remota de aplicações e de protocolos. Para conseguir esta funcionalidade, o Impala impõe que os protocolos e aplicações sejam desenvolvidos de forma modelar. No Impala, e contrariamente ao Yggdrasil, várias aplicações e protocolos podem estar instanciados, mas apenas um pode executar num dado momento.

Com a crescente popularidade da computação na berma [15] e arquitecturas associadas, como *fog* [9] e IoT [1], outras *frameworks* e soluções foram recentemente propostas. Estas incluem Stack4Things [8], e esforços do OpenFog Consortium[2], como a *framework* ENORM [19]. No entanto nenhuma destas soluções se foca em suportar a implementação de protocolos e aplicações para redes ad hoc.

7 Conclusão

Neste artigo apresentamos uma *framework*, denominada Yggdrasil, desenvolvida para simplificar a implementação e desenvolvimento de protocolos e aplicações distribuídas que comunicam através de canais sem fios, numa rede ad hoc. Desenvolvemos um protótipo do Yggdrasil que foi testado extensivamente com o desenvolvimento de protocolos e demonstramos neste artigo o seu uso através de um caso de estudo: o protocolo de encaminhamento para redes ad hoc B.A.T.M.A.N..

[2] https://www.openfogconsortium.org/research/

Pretendemos continuar a desenvolver o Yggdrasil, implementando novas funcionalidades que suportem o desenvolvimento de aplicações recorrendo ao uso de protocolos dinâmicos.

Referências

1. Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., Ayyash, M.: Internet of things: A survey on enabling technologies, protocols, and applications. IEEE Comm. Survs. Tut. **17**(4), 2347–2376 (2015)
2. Cisco: Cisco visual networking index: Global mobile data traffic forecast update. https://tinyurl.com/zzo6766 (2016)
3. Gunaratna, G., Jayarathna, P., Sandamini, S., Silva, D.: Implementing wireless adhoc networks for disaster relief communication. In: Proc. of UMEDIA (2015)
4. Johnson, D., Ntlatlapa, N., Aichele, C.: Simple pragmatic approach to mesh routing using batman. In: Proc. of WCITD. IFIP (2008)
5. Leitão, J., Costa, P.Á., Gomes, M.C., Preguiça, N.: Towards enabling novel edge-enabled applications. arXiv preprint arXiv:1805.06989 (2018)
6. Levis, P., et. al.: TinyOS: An Operating System for Sensor Networks, pp. 115–148. Springer Berlin Heidelberg (2005)
7. Liu, T., Martonosi, M.: Impala: A middleware system for managing autonomic, parallel sensor systems. SIGPLAN Not. **38**(10), 107–118 (2003)
8. Longo, F., Bruneo, D., Distefano, S., Merlino, G., Puliafito, A.: Stack4things: An openstack-based framework for iot. In: 2015 3rd International Conference on Future Internet of Things and Cloud. pp. 204–211 (Aug 2015)
9. Mahmud, R., Kotagiri, R., Buyya, R.: Fog computing: A taxonomy, survey and future directions (2018)
10. Miranda, H., Leggio, S., Rodrigues, L., Raatikainen, K.: A power-aware broadcasting algorithm. In: Proc. of PIMRC. pp. 1–5 (2006)
11. Miranda, H., Pinto, A., Rodrigues, L.: Appia, a flexible protocol kernel supporting multiple coordinated channels. In: Proc. of ICDCS. pp. 707–710 (2001)
12. Moniz, H., Neves, N.F., Correia, M.: Turquois: Byzantine consensus in wireless ad hoc networks. In: Proc. of DSN'10. pp. 537–546 (2010)
13. Neumann, A., Aichele, C., Lindner, M., Wunderlich, S.: Better Approach To Mobile Ad-hoc Networking (B.A.T.M.A.N.). Internet-draft, Int. Eng. Task Force (2008)
14. Riley, G.F., Henderson, T.R.: The ns-3 Network Simulator, pp. 15–34. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
15. Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: Vision and challenges. IEEE IoT Journal **3**(5), 637–646 (2016)
16. Silva, J.a.A., Leitão, J.a., Preguiça, N., Lourenço, J.a.M., Paulino, H.: Towards the opportunistic combination of mobile ad-hoc networks with infrastructure access. In: Proc. of MECC '16. pp. 3:1–3:6. ACM (2016)
17. Simon, G., et. al.: Sensor network-based countersniper system. In: Proc. of SenSys. pp. 1–12. ACM, Baltimore, MD, USA (2004)
18. Torrent-Moreno, M., et. al: Ieee 802.11-based one-hop broadcast communications: Understanding transmission success and failure under different radio propagation environments. In: Proc. of MSWiM '06. pp. 68–77. ACM (2006)
19. Wang, N., Varghese, B., Matthaiou, M., Nikolopoulos, D.S.: Enorm: A framework for edge node resource management. IEEE Transactions on Services Computing pp. 1–1 (2017). https://doi.org/10.1109/TSC.2017.2753775