Enabling Wireless Ad Hoc Edge Systems with Yggdrasil*

Pedro Ákos Costa
NOVA LINCS & DI/FCT/NOVA
University of Lisbon
Lisboa, Portugal
pah.costa@campus.fct.unl

André Rosa
NOVA LINCS & DI/FCT/NOVA
University of Lisbon
Lisboa, Portugal
af.rosa@campus.fct.unl.pt

João Leitão
NOVA LINCS & DI/FCT/NOVA
University of Lisbon
Lisboa, Portugal
jc.leitao@fct.unl.pt

ABSTRACT

Wireless ad hoc networks were extensively studied in the past given their potential for scalability, ease of deployment, and suitability for scenarios where no infrastructure is available. Considering the recent relevance of applications, particularly in the Internet of Things (IoT) and edge computing domains, revisiting these networks becomes a necessity, as to develop novel distributed applications. Distributed applications are highly complex as they require multiple services and abstractions supported by a wide range of distributed protocols, specially in such adverse domains.

To simplify the development of applications in ad hoc networks, in this paper we present Yggdrasil, a novel framework and middleware specifically tailored for the development and execution of distributed applications and associated protocols using commodity devices in such networks. Yggdrasil provides a simple yet effective development environment, which is achieved by combining an event driven programming model with a multi-threaded execution environment that shield the programmer from concurrency issues. A fully functional prototype was developed in C and experimentally evaluated using a fleet of 24 Raspberry Pis.

CCS CONCEPTS

• **Networks** → **Ad hoc networks**; • **Computing methodologies** → **Distributed algorithms**; • **Software and its engineering** → Application specific development environments;

KEYWORDS

Wireless Ad Hoc Networks, Distributed Protocols, Framework

ACM Reference Format:
Pedro Ákos Costa, André Rosa, and João Leitão. 2020. Enabling Wireless Ad Hoc Edge Systems with Yggdrasil. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3341105.3373908

*The work presented here was partially supported by the Lightkone European H2020 Project (under grant number 732505), NOVA LINCS (through the FC&T grant UID/CEC/04516/2013), and NG-STORAGE (through the FC&T grant PTDC/CCI-INF/32038/2017).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SAC '20, March 30-April 3, 2020, Brno, Czech Republic
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3373908

1 INTRODUCTION

Wireless ad hoc networks were extensively studied in the past given their potential for scalability, ease of deployment, and suitability for scenarios where there is no infrastructure, such as war scenarios [37], natural disasters [11], among others. In this context, multiple protocols were proposed by the scientific community, from routing [13, 31], data aggregation [5, 20], fault tolerant dissemination [1, 25], and even byzantine agreement protocols [27]. However, the number of practical applications based on wireless ad hoc networks, particularly applications that have large numbers of devices actively interacting and collaborating, is not significant nowadays, with the exception of very few success cases [2, 21].

The recent increase in popularity and interest on applications in the Internet of Things (IoT) [40] and smart cities [4] domains, among others, motivate the need to revisit and address the inherent challenges of wireless ad hoc networks [32, 34]. These applications often rely on cloud-based solutions however, given the rapid increase in the volume of data being generated by such applications, the cloud is rapidly becoming unable to receive, process, and respond, in a timely fashion, to the generated load. Given that data generated by such applications is expected to continue increasing [6], it is essential to build novel solutions to decentralize data processing. This is also relevant to lower operational costs of such applications and allow them to be easily deployed and operated (potentially by common users).

The edge computing paradigm [35] presents itself as a viable solution, as it promotes moving computations and storage beyond the boundaries of data centers, allowing computations to be performed closer to end clients, potentially (but not exclusively) directly on client devices. In some particular scenarios, such as smart cities, smart spaces, smart agriculture, and smart extraction of resources (e.g., mining), it may be convenient to support the direct interaction between edge devices that lack access to network infrastructures. In this case, the use of wireless ad hoc networks and a variety of distributed protocols for these scenarios, becomes a viable and interesting approach. Additionally, the development of novel protocols and solutions for ad hoc networks, may enable novel distributed applications with edge components, particularly mobile and highly interactive user-facing applications [14, 36].

To enable novel edge applications (that operate over wireless ad hoc networks) to emerge, in this paper we present Yggdrasil, a novel framework and middleware runtime tailored for the development and execution of distributed applications and associated protocols in wireless ad hoc environments. Yggdrasil offers developers a simple and effective development environment that promotes modularity. This is achieved by the clever combination and integration of a set of well known abstractions and techniques that include: an event-driven execution model; a multi-threaded execution environment that shields developers from handling (complex) concurrency issues; simple and intuitive interaction mechanisms between protocols and applications; and low-level communication primitives tailored for wireless ad hoc networks. A prototype of Yggdrasil was developed in C, allowing the execution of protocols and applications in commodity devices (e.g., laptops, micro-computers), while also supporting more resource constrained devices.

The remainder of the paper is organized as follows: Section 2 discusses our system model; Section 3 presents the Yggdrasil framework and its main components; Section 4 illustrates the use of the API provided by Yggdrasil; Section 5 details our experimental work using a fleet of 24 Raspberry Pi 3 - Model B; Section 6 briefly discusses related work; and finally, Section 7 concludes the paper.

2 SYSTEM MODEL

Distributed applications leverage the computational resources of various devices to provide services for either other applications or users. However, achieving this in a distributed setting is a daunting task, more so in wireless ad hoc networks where network infrastructure is not available. Usually, distributed applications rely on a set of distributed protocols that cooperate to provide abstractions that support the operation of the application [10].

In more detail, we model a distributed application for wireless ad hoc networks as a system operating on a wireless ad hoc network, in which a set of processes, that contain application logic and a set of local instances of distributed protocols, cooperate and interact to materialize the application. Each process resides in a node (i.e., device), which has its own resources (CPU, memory, disk, etc). All interactions among different processes are performed through the exchange of messages via the wireless medium and only instances that reside in nodes that are directly reachable through their radios can communicate directly. We say that such processes (or nodes) are neighbors in the ad hoc network.

In the following we analyze a distributed protocol that provides an application with the membership of the system, identifying the primitives that are necessary for a programming framework and runtime middleware (such as Yggdrasil) to support the development of such distributed protocols and applications.

A Simple Membership Protocol. Distributed protocols are often described as algorithms that follow a state machine model [7, 10, 15, 16]. As such, protocols are modeled as having an internal state that can be exposed or evolve through the reception, and handling, of asynchronous events. Algorithm 1 illustrates a simple membership protocol for wireless ad hoc networks that maintains, locally at each node, a set containing the identifiers of all other known processes in the system. The protocol operates by having each node in the system, periodically, sending (through **one-hop broadcast**) a message containing a random sample of up to k nodes present in the local *membership set*. Nodes that receive such a message from another node integrate previously unknown node identifiers into their local state.

The algorithm has a header that denotes the interface of the protocol i.e., the interactions that the protocol supports with other (local) protocols or applications. In this example the protocol is able to process a **request** *getMembership* to which it produces a

Algorithm 1: Example Membership Protocol

```
Interface:
    Requests:
      getMembership ( )
    Replies:
      members ( $m$ ) //m is the set of all known nodes in the system
    Notifications:
      newMember ( $n$ ) //n is the newly found member of the system

1: Local State:
2:     membership //The set of members in the system
3:     $k$ //sample size

4: Upon Init ( $pid$, $\Delta T$ , $sampleSize$) do:
5:     membership ⟵ $\{pid\}$ //pid is the local process identifier
6:     $k \longleftarrow sampleSize$
7:     Setup Periodic Timer Announce ($\Delta T$)

8: Upon Timer Announce do: //every $\Delta T$
9:     subSet ⟵ getMemberSubSet($membership$, $k$)
10:    Trigger OneHopBCast SAMPLE ( $subSet$ )

12: Upon Receive SAMPLE ( $subSet$ ) do:
13:    foreach $n \in subSet : n \notin$ membership do:
14:      membership ⟵ membership $\cup$ $n$
15:      Trigger Notification newMember ( $n$ )

16: Upon Request getMembership ( ) do:
17:    Trigger Reply members ( $membership$ )
```

reply *members* containing a copy of the locally maintained set of node identifiers. Furthermore, the membership protocol denoted in Algorithm 1 also issues a **notification** *newMember* whenever a new identifier is added to the local set. This notification is delivered to any system component (protocol or application) that registers interest in receiving it. Notice that while a reply is always generated in response to a request (and directed to the original issuer of the request), a notification can be generated at any point in the execution of a protocol.

The algorithm defines the local state maintained by the protocol in each node (Alg. 1 lines 1 – 3). The protocol maintains the set of known members in the system (Alg. 1 line 2), and the k parameter that controls the maximum sample size sent in periodic messages. Following the state machine nomenclature, a special **Init** handler is used to initialize the protocol internal state upon bootstrap. In the case of this membership protocol, this is achieved by initializing the membership set with the local node identifier, and setting the value for parameter k (Alg. 1 lines 5 – 6). Since the algorithm relies on a periodic action (sending a message) it also configures a **periodic timer** (that triggers every ΔT). The reception of this special Timer event, leads the protocol state machine to execute the transmission of the *sample message* containing the sample of the local membership set (Alg. 1 lines 8 – 10).

When a sample message is received (Alg. 1 line 12), the protocol adds the unknown members contained in the message to its local membership (Alg. 1 lines 13 – 14) and triggers a notification to be delivered to any interested local protocol or application, that notifies the existence of a new node in the system (Alg. 1 line 15).

3 YGGDRASIL

Yggdrasil is a programming framework and middleware runtime that allows distributed protocols and applications to be easily developed following their specification. This is possible by the clear and specifically tailored programming model exposed by Yggdrasil.

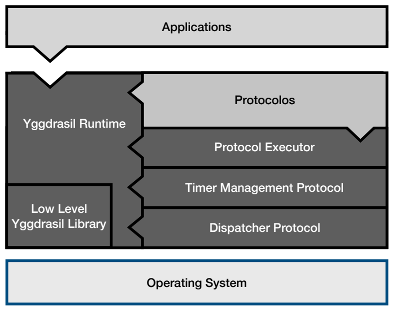


Figure 1: Simplified Yggdrasil's Architecture.

Furthermore, at its core, Yggdrasil combines multiple techniques to ensure efficient execution of protocols and applications, namely by ensuring parallel execution (when the hardware allows it) and ensuring low memory and processing footprints, making Yggdrasil suitable not only for commodity devices but also to devices with limited capability.

3.1 Protocols in Yggdrasil

In Yggdrasil, each protocol is modeled (and implemented) as a state machine. In more detail, each protocol has its own internal state that evolves accordingly to the (sequential) reception and processing of events. Protocols can generate events to be processed by themselves, or to be delivered to other protocols. The events that guide the execution of protocols are divided in four types, motivated by common distributed protocols and applications operation and interaction patterns. These are: *i*) ***Messages***, that are the only type of event that can be transported between processes (i.e., different nodes), messages can be destined to all neighboring nodes (i.e., one-hop broadcast) or to a single neighbor (i.e., point-to-point); *ii*) ***Timers***, that notify the execution of some periodic task or that a local timeout occurred; *iii*) ***Requests/Replies***, that allow the direct one-to-one interaction between protocols in the local process; and finally, *iv*) ***Notifications***, that allow the indirect one-to-many interaction between protocols in the local process.

Within a Yggdrasil process each protocol and application is, by default, executed in the context of an independent execution thread, enabling their evolution in parallel. Protocols (and applications) are provided with an *event queue* from which they wait for events to be consumed. Each protocol and application is internally associated with a *numeric identifier*. Events are tagged with the numeric identifier of the protocol which created the event and to which it should be delivered. This allows events created by one protocol to be delivered by Yggdrasil to another protocol (or application) by pushing the event into the appropriate event queue.

We note however, that certain event types are more delay sensitivity than others (e.g., Timers). To cope with this, event queues are modeled to prioritize event types, meaning that when different event types are present in the queue, those are consumed in a specific order according to their type: first timers, then notifications, messages, and finally requests/replies.

3.2 Yggdrasil Design

Figure 1 depicts a high-level overview of Yggdrasil's architecture. Yggdrasil operates above the operating system, in particular, it considers a unix-based general-purpose operating system. From the operating system we assume abstractions to configure the radio device and the wireless network, in addition to standard abstractions and programming interfaces (e.g., concurrent execution, synchronization mechanisms such as mutex and semaphores).

Yggdrasil is composed by four main components (darker toned components in Figure 1) that cooperate to support the execution of protocols and applications. These include: the **Yggdrasil Runtime**, which configures the radio device to enable network communication through the **Low Level Yggdrasil Library**, and handles the execution of protocols and applications by providing the API and managing event queues to consume and produce events; a **Dispatcher Protocol**, that handles all network communication; the **Timer Management Protocol**, which monitors all Timer events set within the local Yggdrasil process; and a **Protocol Executor**, that allows some protocols to share a single execution thread.

In more detail, an Yggdrasil application starts by initializing the Yggdrasil Runtime, providing it with a network configuration which includes the radio mode (in this case ad hoc), the frequency of the radio, and the network name to be joined (or created). The Runtime configures the radio device through the Low Level Yggdrasil Library, which uses low level system calls to manipulate the radio interface. This library is also responsible to provide the Runtime with a *Channel* abstraction, through which network messages are exchanged. The channel is configured as a low-level network communication device, that allows messages to be exchanged directly at the MAC layer (i.e., layer 2 of the Network Stack). A kernel packet filter is also installed in the channel to filter unwanted network messages from sources other than Yggdrasil processes (e.g., ARP packets).

Additionally, the Runtime prepares the remaining main components of Yggdrasil, which we name *core protocols*. This entails initializing an event queue for each core protocol and providing the Dispatcher access to the previously created channel. Core protocols have reserved numeric identifiers however, the application developer can override each one with a variant of her own design providing similar or enriched functionality. For instance, the Dispatcher Protocol could be replaced by one that encrypts and decrypts all network messages.

After the Runtime is initialized, the application registers itself and the protocols it will use in the Runtime. The Runtime prepares and associates an event queue for each protocol and application, and triggers the special Init event for each protocol and application (which in turn, initializes their internal state). Each protocol and application contains a configuration that is provided to the Runtime upon their registry. This configuration includes information about produced and consumed notifications (the latter from other protocols).

Execution Modes. Each protocol and application configuration also states how it should be handled by the Yggdrasil Runtime: it can rely on an independent execution thread; or share a single execution thread with other protocols. The latter is achieved by taking advantage of the Protocol Executor.

Protocols that execute within the Protocol Executor, share the same event queue, being the Executor responsible to multiplex events across protocols. The Protocol Executor also allows protocols to be safely started and stoped at runtime. This allows to have protocols that can be dynamically used by applications to deal with changes on the operation environment (e.g., using protocols that make less use of the wireless medium, if the wireless medium appears to be saturated).

Notice that in Yggdrasil the programmer is not responsible to manage the execution threads of any protocol. Instead these are managed internally by the Yggdrasil Runtime.

Support for Piggybacking. In some cases a protocol might benefit from piggybacking information on messages sent by other protocol. Piggybacking information is a technique often employed to perform optimizations, in protocols and applications, to lower the amount of messages sent to the network. This is particularly relevant when information to be sent is small, since it allows to lower the occupation of the wireless medium. To enable this behavior, the Yggdrasil Runtime allows a protocol to intercept the event queue of another protocol. This will lead the Yggdrasil Runtime to transparently deliver events destined to the intercepted protocol to the interceptor instead. As pointed out above, this is particularly useful to add control information to messages. However, the interceptor protocol, becomes responsible for: *i*) routing the event to the original destination (for messages, usually the Dispatcher), and *ii*) replace the destination of the message to be itself, such that on the remote node, any control information added can be removed, before the message reaching its original destination.

4 IMPLEMENTING OUR SIMPLE PROTOCOL

A fully functional prototype of Yggdrasil was implemented using the C language, and has been used to implement various protocols and simple (demo) applications that exercise these protocols. These include, neighbor discovery, fault detectors, different flavors of distributed aggregation, application-level broadcast, routing, reliable point-to-point communication, topology control, and experiment management protocols; the last two are dedicated to simplify the execution of (distributed) experimental evaluations. The prototype is available at https://github.com/LightKone/Yggdrasil.

To showcase how a protocol can be implemented in Yggdrasil, we present an implementation (Figure 2) of the membership protocol specified in Algorithm 1. Similar to the specification, the implementation is divided in three parts: *i*) the local state, *ii*) the event handlers, and *iii*) the initialization of the protocol. The header section presented in Algorithm 1 is omitted due to lack of space (this is reported to Yggdrasil as part of the protocol configuration).

We note that there is an expansion in the number of lines between the C implementation and the pseudo-code represented in Algorithm 1 however, given the verbosity of the C language, we consider the implementation to be quite compact, specially when comparing an implementation leveraging Yggdrasil and a C standalone implementation of the same protocol.

Local State. In Yggdrasil a protocol defines a structure containing the state it maintains. Lines 2 – 3 in Fig. 2 directly map to lines 3 – 4 in Alg. 1. An `uuid_t` is a universally unique identifier, represented by a `char[16]` and a `list*` is a pointer to a generic list type. The following line 5 in Fig. 2 is a variable specific to Yggdrasil, the `proto_id`, which represents the protocol's unique numeric identifier, whose value should be defined (by Yggdrasil convention) in the protocol's header file.

Event Handlers. The protocol handles three types of events: timers (Fig. 2 lines 8 – 18); messages (Fig 2 lines 20 – 33); and requests

```
1  /* ---------------- Local State ------------------ */
2  typedef struct state {
3    list* members;
4    int k;
5    short proto_id;
6  };
7  /* --------------- Event Handlers ---------------- */
8  static short Announce(YggTimer* timer, struct state* state) {
9    YggMessage msg; YggMessage_initBcast(&msg, state->proto_id);
10   list* subSet = getMemberSubSet(state->members, state->k);
11   while(subSet->size > 0) {
12     char* member_id = list_remove_head(subSet);
13     YggMessage_addPayload(&msg, member_id, sizeof(uuid_t));
14     free(member_id);
15   }
16   dispatch(&msg); YggMessage_freePayload(&msg);
17   return SUCCESS;
18 }
19
20 static short ReceiveSample(YggMessage* msg, struct state* state) {
21   void* ptr = NULL; uuid_t id;
22   while((ptr = YggMessage_readPayload(msg, ptr, id, sizeof(uuid_t))) != NULL) {
23     if(list_find_item(state->members, (equal_function) equal_id, id) == NULL){
24       char* mid = malloc(sizeof(uuid_t)); memcpy(mid, id, sizeof(uuid_t));
25       list_add_item_to_head(state->members, mid);
26
27       YggEvent ev; YggEvent_init(&ev, state->proto_id, NEW_MEMBER);
28       YggEvent_addPayload(&ev, id, sizeof(uuid_t));
29       deliverEvent(&ev); YggEvent_freePayload(&ev);
30     }
31   }
32   return SUCCESS;
33 }
34
35 static short getMembers(YggRequest* req, struct state* state) {
36   if(req->request == REQUEST && req->request_type == GET_MEMBERS){
37     YggRequest_freePayload(req); //should already be NULL
38     YggRequest_Reply(req, state->proto_id);
39     for(list_item* i = state->members->head; i != NULL; i = i->next) {
40       YggRequest_addPayload(req, (char*) i->data, sizeof(uuid_t));
41     }
42     deliverReply(req); YggRequest_freePayload(req);
43     return SUCCESS;
44   }
45   return FAILED;
46 }
47 /* ---------------- Initialization -----------------  */
48 proto_def* membership_init(membership_param* args) {
49   struct state* state = malloc(sizeof(struct state));
50   char* pid = malloc(sizeof(uuid_t); getmyId(pid);
51   state->members = list_init(); list_add_item_to_head(state->members, pid);
52   state->proto_id = PROTO_MEMBERSHIP;
53
54   proto_def* membership = create_protocol_definition(state->proto_id,
         "Membership", state, membership_state_destroy);
55   proto_def_add_produced_notifications(membership, 1); //NEW_MEMBER
56
57   proto_def_add_msg_handler(membership, ReceiveSample);
58   proto_def_add_timer_handler(membership, Announce);
59   proto_def_add_request_handler(membership, getMembers);
60
61   YggTimer_init(&state->announce, state->proto_id, state->proto_id);
62   YggTimer_set(&state->announce,
63     args->announce_period_s, args->announce_period_ns, //first notification
64     args->announce_period_s, args->announce_period_ns);//periodicity
65
66   setupTimer(&state->announce);
67   return membership;
68 }
```

Figure 2: Membership Yggdrasil Implementation.

(Fig. 2 lines 35 – 46). We note however, that if a protocol processes different events of the same type (e.g., different types of messages), these have to be multiplexed by the protocol when handling that type of event. This is because the Yggdrasil Runtime does not have knowledge of how protocols internally operate to avoid breaking the isolation that Yggdrasil imposes over protocols, and minimize complex interactions between protocols and the Yggdrasil Runtime, such as registering each message handled by a protocol individually.

When the protocol processes a timer (Fig. 2 line 8), it begins by creating and initializing a message to be sent to the MAC broadcast address (`ff:ff:ff:ff:ff:ff`) (Fig. 2 line 9). The protocol proceeds to generate a subset of members (we omit details on this function) and adds each node identifier to the message's payload (Fig. 2 lines 10–15). The instruction `YggMessage_addPayload`, copies the contents of `state->n_id` with size `sizeof(uuid_t)`, to the next available position in the message's payload (calculated automatically by taking into consideration the number of bytes added previously) and updates the size of the message payload. The message is sent to the Dispatcher with the instruction `dispatch` (Fig. 2 line 16), which performs a deep copy into the Dispatcher Protocol's event queue. Deep copies are used when adding events to queues to avoid concurrency issues. Finally, because event payloads are dynamically allocated in Yggdrasil, the payload of the message must be freed (Fig. 2 line 16).

When the protocol processes a message (Fig. 2 line 20), it reads the member identifiers in the payload of the received message, and verifies, for each one, if it is already contained in its local membership list. If a new member is found (not in the local membership list), the protocol adds the new member's identifier to its local membership list (Fig. 2 lines 22 – 25). The important aspect to notice here is how the payload is read with the instruction `YggMessage_readPayload`. This instruction reads `sizeof(uuid_t)` bytes from the payload of the `msg` starting at the position pointed by `ptr` (if `NULL`, from the beginning), stores the read contents in `id`, and returns a pointer to the last position read (or `NULL` if there are no more bytes to be read). The protocol proceeds to create a notification (defined as `YggEvent`), adds the new member's id to the notification payload, asks the Runtime to deliver it to all interested protocols and applications, and frees the payload (Fig. 2 lines 27 – 29).

Lastly, when the protocol processes a request (Fig. 2 line 35), it first certifies that the received request is valid (according to the protocol logic), prepares the reply (adding all known members to the reply payload) and delivers the reply to the corresponding protocol, freeing the payload in the end (Fig. 2 lines 36 – 42). This is analogous to how previous events are handled.

Initialization. The initialization of the membership protocol (Fig. 2 line 48) is the one that differs the most from Algorithm 1. This is because this function is also responsible to define the protocol configuration for the Yggdrasil Runtime. We named this configuration: *protocol definition* (or `proto_def`). The functions begins by initializing the protocol state (Fig. 2 lines 49 – 52). It then creates a protocol definition, that contains the protocol's numeric identifier, its name, state, and a function to release the state (Fig. 2 line 54). Finally, it defines the notifications produced by this protocol, that can be consumed by any other protocol or applications (Fig. 2 line 55).

The function proceeds to register event handlers that the protocol has (Fig. 2 lines 57 – 59). This will signal the Yggdrasil Runtime to register the protocol in the Protocol Executor. Alternatively, the protocol can register a *main loop function* to signal the desire to execute in an independent execution thread. The main loop function of a protocol will be provided with the protocol's event queue and state, and is responsible to retrieve events from the event queue and multiplex them by type, calling the corresponding event handlers. This provides more flexibility to the developer to implement complex behaviors in her protocol. We omit the implementation of this function due to lack of space.

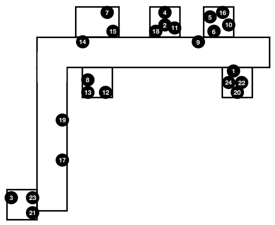


Figure 3: Experimental Configuration.

Finally, the initialization proceeds to setup the announce timer event for its periodic task (Fig. 2 lines 61 – 66); and returns the protocol definition to be processed by the Yggdrasil Runtime.

5 EXPERIMENTAL EVALUATION

In this section we present our experimental evaluation of Yggdrasil to complement and demonstrate that distributed protocols and applications can be implemented and executed within Yggdrasil with minimal effort and performance overhead. The evaluation is divided in two parts: In the first, we experimentally evaluate implementations of three different classes of distributed protocols and simple demo applications that exercise them using Yggdrasil. In the second, we measure the overhead of events (in particular messages) passing through the event queues of Yggdrasil.

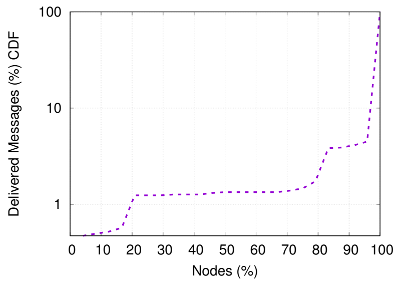
5.1 Protocols Performance in Yggdrasil

We have implemented three classes of protocols for wireless ad hoc networks (and demo applications that exercise them) in Yggdrasil. These include: *i*) a simple broadcast protocol based on flooding (similar to the one discussed in [16] for wired networks); *ii*) a popular routing protocol for wireless ad hoc networks, the B.A.T.M.A.N. (V4) routing protocol [13]; and *iii*) an aggregation protocol, GAP [8].

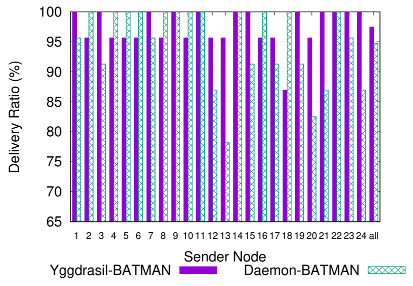
The Yggdrasil demo applications were deployed in a real testbed composed by a fleet of 24 Raspberry Pis 3 - Model B, executed for a period of 10 minutes, and evaluated relevant performance metrics for each protocol class. The devices were scattered through our department building across two hallways (each with approximately 30*m*) and rooms as depicted in Figure 3. We also note that some devices were positioned near access points and other devices frequently polluting the wireless environment. Each demo application produces logs related to the protocol it is exercising, which were post-processed after all experiences had concluded. In the following we briefly describe each application and protocol and discuss the results in Figure 4.

Broadcast Protocol. The application that exercises the broadcast protocol operates as follows: every two seconds, each process independently and randomly decides with a probability of 50% to broadcast a message containing the identifier of the process and a unique monotonic identifier by issuing a request to the broadcast protocol. All nodes register to a log messages disseminated by them and messages received from the network.

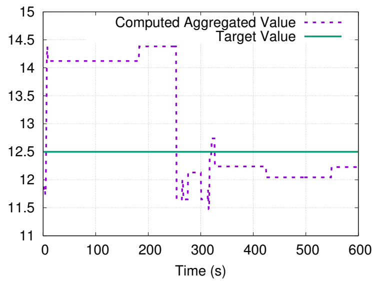
The broadcast protocol is fairly simple. It first retrieves the message to be disseminated from the request, delivers it to the application, and schedules the message for transmission with a jitter, as to avoid broadcast storms [30]. Messages delivered are stored in a list to ensure at most once delivery semantics and garbage collected



(a) Broadcast Protocol: Delivery Ratio (CDF).



(b) Routing Protocol: Comparison of Delivery Ratio per Node.



(c) Aggregation Protocol: Precision.

Figure 4: Performance Evaluation of Distributed Protocols.

(after a long enough period of time) by a periodic task. The protocol is implemented with approximately 200 lines of C code.

Performance Results. In this experiment we consider as performance metric the delivery rate of broadcast messages (i.e., the fraction of nodes that deliver a given message). Figure 4a reports our results in the form of a commutative distribution function (CDF) that shows the percentage of messages (in the y-axis, note that it is in logarithmic scale) in function of the fraction of nodes that delivered it (in the x-axis). The results show that the large majority of disseminated messages (in total there were approximately 3, 500 messages disseminated) were delivered by every node. Only a small fraction, below 10%, of messages were delivered by fewer nodes. This however is not surprising, since collisions in the wireless medium still happen, despite the transmission delays employed by the protocol to mitigate this effect.

Routing Protocol. The application that exercises the routing protocol operates akin to the previous one. It decides to transmit a message with a probability of 50% every two seconds, to a randomly chosen node out of the 23 possible destinations (nodes do not send messages to themselves) and requests the B.A.T.M.A.N. protocol to deliver it to the correct destination. A variant of this application was also used to exercise a reference implementation of the B.A.T.M.A.N. (also written in C) that operates as a linux daemon (found in https://www.open-mesh.org/projects/open-mesh/wiki). In both applications, each node logs all messages sent and received.

The operation and specification of the B.A.T.M.A.N. protocol can be found in [29]. Our implementation performs some simplifications regarding the original specification, which do not compromise the operation of the protocol. In particular, we represent the protocol's sliding window (used to compute routes) in an array of shorts, rather than a bit mask; and ignore aspects to support gateways and external networks. This totals in less than 500 lines of C code. An informal comparison with the reference implementation reveals, after a careful analysis of the code, that the reference implementation represents the core logic of the protocol in almost 4 times more lines of code (roughly 2, 000 lines) than our own. This is mostly due to the daemon having to deal with every aspect of the protocol's operation, from the network interface management to the serialization of messages into network buffers.

Performance Results. In this experimental comparison we measure the fraction of messages sent by each node that were effectively received at their destination. Figure 4b reports the delivery ratio per individual node in our deployment, using each of the routing alternatives, as well as the average delivery ratio for all nodes (the final pair of columns labeled *all*). We note that in each experiment there were approximately 3, 500 messages sent, with each node transmitting close to 150 messages. The results show that both implementations present very similar performance, validating that Yggdrasil can simplify the development of distributed protocols. In fact, when considering the average delivery ratio, our implementation surpasses the B.A.T.M.A.N. daemon by 3%. We note that this difference is most probably caused by wireless interference. Note that both implementations of the protocol route messages using unicast messages, which benefit from collision-avoidance and retransmission mechanisms implemented at the MAC layer, which justifies why routing is less affected by collisions in the wireless medium when compared with the broadcast protocol.

Aggregation Protocol. The application that exercises the aggregation protocol periodically (every second) queries GAP [8] for the estimate of the aggregation result. GAP is configured to compute the average aggregate function. The input values for the aggregation are statically configured to be each node's identifier (a number from 1 to 24), which implies that the correct computed average value is 12.5. GAP is configured to transmit periodic updates of its local estimate of the aggregate value (and other information related to the protocol's operation) every two seconds (while the estimate is not considered stable). GAP requires a node to act as a root node for a tree established by the protocol. A random node was selected for this purpose (node 5). Furthermore, the application also configures a discovery protocol enriched with a fault detection mechanism to provide GAP with the local neighborhood.

More details on the operation of GAP can be found in the original paper [8]. Our implementation follows the complete protocol as specified by the authors. However, the authors propose different policies to deal with the management of estimates received from neighbors. Our implementation only supports the proposed default policy, which maintains all estimates from neighbors as long as they are not suspected of failure. The protocol was implemented in less than 400 lines of C code.

Performance Results. Figure 4c reports the obtained results, only at the root node, where we depict the estimate of the aggregated value over time (we remind the reader that all experiments were conducted for a period of 10 minutes). For the convenience of the reader we also present a green solid line that represents the

Table 1: 95^{th} percentile delay of sending a message in milliseconds.

	A:W/out Yggdrasil	**B:**W/Yggdrasil	**C:**W/Idle	**D:**W/Idle Intercept
Raspberry Pi	0.110729	0.182161	0.186275	0.211146
GRiSP	1.983731	3.137988	3.155992	3.418891

target (i.e., correct) value. This allows to infer the precision of the aggregation process in the root node. The results show that for the first few hundreds of seconds of the experiment, the root node has an estimate that only deviates from the correct value in approximately 1.7 units. This value then fluctuates around the 250 seconds mark and get much closer to the correct value (a difference as small as 0.25 units) for the remainder of the experiment. The protocol is unable to achieve the correct value, due to frequent loss of messages in a segment of the ad hoc network. This clearly shows the practical benefits of employing tools such as Yggdrasil for validation of solutions in ad hoc networks, that allows to run experiments on real settings instead of relying solely on simulation [12, 33].

5.2 Overhead Evaluation

To measure the overhead generated by the use of Yggdrasil, we have conducted an experiment where we measured the amount of time required for a message to be sent to the network (i.e., request the kernel to send the message to the network) in several configurations. In the following we detail our experimental methodology and obtained results.

Experimental Methodology. In this experiment we have developed four variants of a simple application in C code, that sends a message to the network every second. This application is executed for more than 10, 000 seconds (slightly bellow 3 hours), sending a total of 10, 000 messages. The application variants are the following: *i*) in variant ***A*** the application is not implemented using Yggdrasil, as such it sends messages by directly using the OS kernel; *ii*) in variant ***B*** the application is implemented using Yggdrasil. The application delegates the functionality of sending the message to the Dispatcher Protocol; *iii*) variant ***C*** is similar to the previous having the application executing concurrently with an idle protocol; and *iv*) variant ***D:*** the idle protocol intercepts the Dispatcher protocol's event queue processing all messages sent by the application.

These experiments were executed in one of the Raspberry Pis, which has a four core CPU with a clock rate of 1.2*Ghz* and 1*GB* of memory; and on an embedded system board named GRiSP in its first version (https://www.grisp.org), having a microchip CPU with a clock rate of 300*Mhz* and 64*MB* of memory. This board executes a Real Time Embedded Multiprocessor System (RTEMS), which is based on an unix system that is directly linked (as a collection of libraries) with the application to be executed in the board (also implemented in C). For each variant we logged a timestamp immediately before the creation of the message, and immediately after the message was delivered to the network. We correlated these data points for each message offline. During each experiment, the CPU and memory consumptions were also gathered using standard profiling tools (e.g., perf, top, and built-in tools in RTEMS).

Experimental Results. Table 1 reports the 95^{th} percentile delay of sending a message in milliseconds for each of our experiments detailed above. The results show that employing Yggdrasil (in variant ***B***) in a Raspberry Pi incurs in an increased delay of around 0.07 milliseconds, whereas in a GRiSP board, the increased delay is around 1.15 milliseconds. The overhead increase is expected, since additional processing is required.

Executing another protocol concurrently (variant ***C***) results in a relatively small overhead that is not significant (0.004 and 0.02 milliseconds in the Raspberry Pi and the GRiSP respectively). When the protocol intercepts the message (variant ***D***) it results in an increased delay of 0.025 milliseconds in a Raspberry Pi and 0.35 milliseconds in a GRiSP. The overhead is caused by the intercept behavior requiring two additional memory copies for the message to reach the Dispatcher.

Regarding the CPU and memory consumptions we have observed few variations of values in our experiments. The CPU consumption in the Raspberry Pi was observed to be around 0.01%, while in the GRiSP board it was observed to be approximately 1%. The low CPU usage is expected has most of the components in Yggdrasil are idly waiting for events (i.e., not performing active waits). The memory consumption was observed to be in both platforms approximately 700 KiB (1 KiB = 1024 bytes) of resident memory (this value does not take into account the memory footprint of RTEMS in the GRiSP board). This implies that the memory footprint of Yggdrasil is relatively low (bellow 1 MB).

Overall, our results show that Yggdrasil presents a very modest overhead considering the functionalities it provides to protocols and applications and the gains in terms of implementation effort for developers.

6 RELATED WORK

There have been other frameworks and middleware solutions proposed to support the development of distributed protocols/systems and their execution. Yggdrasil, to the best of our knowledge, is the first to combine a simple and generic development environment for wireless ad hoc networks with the ability to be executed in general purpose operating systems on commodity devices.

Yggdrasil shares some design principles with previously proposed protocol composition frameworks, in particular, its event-driven programming model. Frameworks, such as Appia [26], Horus [38], Eva [3], and Cactus [24] were original designed considering wired environments, and lack low-level communication primitives that allow processes to easily (without resorting to IPs) exchange messages on a wireless ad hoc network.

Lightweight operating systems, such as TinyOs [17], Impala [18], and Contiki [9], are used to build wireless ad hoc systems. Nevertheless, these specialize in wireless sensor networks, and target small families of hardware, failing to be general purpose.

Middlewares that provide communication among processes in ad hoc networks have also been proposed. These include STEAM [23] and EMMA [28]. However, their focus is solely providing communication primitives based on point-to-point and pub-sub models, instead of a generic programming model.

More distant from the goals of this work, and reacting to the increasing popularity of edge computing [35] and fog computing [22], other frameworks and solutions to support the development of applications in these contexts have recently been proposed, mostly in the field of Internet of Things (IoT) [40]. These include solutions and systems such as Stack4Things [19], and ENORM [39]. Nonetheless, these solutions do not focus on wireless ad hoc networks, nor on building the fundamental support to leverage those networks to perform complex tasks cooperatively.

7 CONCLUSION

In this paper we address the challenges related with developing, implementing, and executing, distributed applications and protocols for wireless ad hoc networks that leverage commodity devices and general purpose operating system, while supporting resource constrained devices. We have presented a novel framework and middleware runtime, named Yggdrasil, that offers key abstractions and a programming model that is highly suitable for developing distributed protocols. We have developed a prototype of Yggdrasil in the C language, and have extensively evaluated it by implementing three different classes of distribute protocols. These include broadcast, routing, and aggregation protocols. We have experimentally validated our implementations and have shown that in general, Yggdrasil provides abstractions that allow to implement these protocols with a lower amount of effort (when compared with stand alone implementations), and that the execution of protocols using Yggdrasil presents adequate performance. We expect Yggdrasil to be an enabler of novel solutions, and assist in developing novel IoT and smart cities/spaces applications.

As future work, we plan to enrich Yggdrasil to support other network abstractions and to improve support to autonomic reconfiguration of individual protocols and protocol stacks in response to dynamic runtime conditions.

REFERENCES

[1] M. Akter, A. Islam, and A. Rahman. 2016. Fault tolerant optimized broadcast for wireless Ad-Hoc networks. In *2016 International Conference on Networking Systems and Security (NSysS)*. 1–9.

[2] C. Ameixieira et al. 2014. Harbornet: a real-world testbed for vehicular networks. *IEEE Communications Magazine* 52, 9 (2014), 108–114.

[3] F. Brasileiro, F. Greve, M. Hurfin, J. . Le Narzul, and F. Tronel. 2001. EVA: an event-based framework for developing specialised communication protocols. In *Proceedings IEEE International Symposium on Network Computing and Applications. NCA 2001.* 108–119.

[4] C. Cabrera, G. White, A. Palade, and S. Clarke. 2018. The Right Service at the Right Place: A Service Model for Smart Cities. In *2018 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. 1–10.

[5] Jen-Yeu Chen, G. Pandurangan, and Dongyan Xu. 2006. Robust Computation of Aggregates in Wireless Sensor Networks: Distributed Randomized Algorithms and Analysis. *IEEE Transactions on Parallel and Distributed Systems* 17, 9 (9 2006), 987–1000.

[6] Cisco. 2016. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update. https://tinyurl.com/zzo6766.

[7] Pedro Ákos Costa and João Leitão. 2018. Practical Continuous Aggregation in Wireless Edge Environments. In *Proc. of 37th IEEE International Symposium on Reliable Distributed Systems (SRDS'18)*. IEEE, Salvador, Brazil.

[8] Mads Dam and Rolf Stadler. 2005. A generic protocol for network state aggregation. In *Proc. Radiovetenskap och Kommunikation (RVK)*. Citeseer, 14–16.

[9] A. Dunkels, B. Gronvall, and T. Voigt. 2004. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks.* 455–462.

[10] Rachid Guerraoui and Luís Rodrigues. 2006. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[11] G. T. C. Gunaratna, P. V. N. M. Jayarathna, S. S. P. Sandamini, and D. S. De Silva. 2015. Implementing wireless Adhoc networks for disaster relief communication. In *2015 8th International Conference on Ubi-Media Computing (UMEDIA)*. 66–71.

[12] Teerawat Issariyakul and Ekram Hossain. 2010. *Introduction to Network Simulator NS2* (1st ed.). Springer Publishing Company, Incorporated.

[13] David Johnson, Ntsibane Ntlatlapa, and Corinna Aichele. 2008. Simple pragmatic approach to mesh routing using BATMAN. In *2nd IFIP International Symposium on Wireless Communications and Information Technology in Developing Countries.*

[14] J. Leitão, P. Á. Costa, M. C. Gomes, and N. Preguiça. 2018. *Towards Enabling Novel Edge-Enabled Applications.* Technical Report. https://arxiv.org/abs/1805.06989

[15] João Leitão, José Pereira, and Luis Rodrigues. 2007. Epidemic broadcast trees. In *Proc. of SRDS'07.* IEEE.

[16] João Leitão, José Pereira, and Luis Rodrigues. 2007. HyParView: A membership protocol for reliable gossip-based broadcast. In *Proc. of DSN'07.* IEEE.

[17] P. Levis et al. 2005. TinyOS: An Operating System for Sensor Networks. In *Ambient Intelligence.* Springer Berlin Heidelberg, Berlin, Heidelberg, 115–148.

[18] Ting Liu and Margaret Martonosi. 2003. Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. *SIGPLAN Not.* 38, 10 (June 2003), 107–118.

[19] F. Longo, D. Bruneo, S. Distefano, G. Merlino, and A. Puliafito. 2015. Stack4Things: An OpenStack-Based Framework for IoT. In *2015 3rd International Conference on Future Internet of Things and Cloud.* 204–211.

[20] Samuel Madden et al. 2002. TAG: A Tiny AGgregation Service for Ad-hoc Sensor Networks. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 131–146.

[21] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. 2005. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)* 30, 1 (2005), 122–173.

[22] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. 2018. Fog Computing: A Taxonomy, Survey and Future Directions. In *Internet of Everything: Algorithms, Methodologies, Technologies and Perspectives.* Springer Singapore, Singapore, 103–130.

[23] R. Meier and V. Cahill. 2002. STEAM: event-based middleware for wireless ad hoc networks. In *Proceedings 22nd International Conference on Distributed Computing Systems Workshops.* 639–644.

[24] S. Mena, X. Cuvellier, C. Gregoire, and A. Schiper. 2003. Appia vs. Cactus: comparing protocol composition frameworks. In *Proc. of SRDS'03.* 189–198.

[25] H. Miranda, S. Leggio, L. Rodrigues, and K. Raatikainen. 2006. A Power-Aware Broadcasting Algorithm. In *2006 IEEE 17th International Symposium on Personal, Indoor and Mobile Radio Communications.* 1–5.

[26] H. Miranda, A. Pinto, and L. Rodrigues. 2001. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc.of ICDCS'01.* 707–710.

[27] H. Moniz, N. F. Neves, and M. Correia. 2013. Byzantine Fault-Tolerant Consensus in Wireless Ad Hoc Networks. *IEEE Transactions on Mobile Computing* 12, 12 (Dec 2013), 2441–2454.

[28] Mirco Musolesi, Cecilia Mascolo, and Stephen Hailes. 2005. EMMA: Epidemic Messaging Middleware for Ad Hoc Networks. *Personal Ubiquitous Comput.* 10, 1 (Dec. 2005), 28–36.

[29] Axel Neumann, Corinna Aichele, Marek Lindner, and Simon Wunderlich. 2008. *Better Approach To Mobile Ad-hoc Networking (B.A.T.M.A.N.).* Internet-Draft draft-openmesh-b-a-t-m-a-n-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-openmesh-b-a-t-m-a-n-00

[30] Sze-Yao Ni, Yu-Chee Tseng, Yuh-Shyan Chen, and Jang-Ping Sheu. 1999. The Broadcast Storm Problem in a Mobile Ad Hoc Network. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '99).* ACM, Seattle, Washington, USA, 151–162.

[31] C. E. Perkins and E. M. Royer. 1999. Ad-hoc on-demand distance vector routing. In *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA '99. Second IEEE Workshop on.* 90–100.

[32] R. Ramanathan and J. Redi. 2002. A brief overview of ad hoc networks: challenges and directions. *IEEE Communications Magazine* 40, 5 (2002), 20–22.

[33] George F. Riley and Thomas R. Henderson. 2010. The ns-3 Network Simulator. In *Modeling and Tools for Network Simulation.* Springer Berlin Heidelberg, Berlin, Heidelberg, 15–34.

[34] Marcelo G. Rubinstein et al. 2006. A Survey on Wireless Ad Hoc Networks. In *Mobile and Wireless Communication Networks.* Springer US, Boston, MA, 1–33.

[35] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (10 2016), 637–646.

[36] João A. Silva, João Leitão, Nuno Preguiça, João M. Lourenço, and Hervé Paulino. 2016. Towards the Opportunistic Combination of Mobile Ad-hoc Networks with Infrastructure Access. In *Proceedings of the 1st Workshop on Middleware for Edge Clouds & Cloudlets (MECC '16).* ACM, New York, NY, USA, Article 3, 6 pages.

[37] Gyula Simon et al. 2004. Sensor Network-based Countersniper System. In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems (SenSys '04).* ACM, Baltimore, USA, 1–12.

[38] Robbert Van Renesse, Takako M. Hickey, and Kenneth P. Birman. 1994. *Design and Performance of Horus: A Lightweight Group Communications System.* Technical Report. Cornell University, Ithaca, NY, USA.

[39] N. Wang, B. Varghese, M. Matthaiou, and D. S. Nikolopoulos. 2017. ENORM: A Framework For Edge NOde Resource Management. *IEEE Transactions on Services Computing* (2017), 1–1.

[40] G. White, A. Palade, C. Cabrera, and S. Clarke. 2018. IoTPredict: Collaborative QoS Prediction in IoT. In *2018 IEEE International Conference on Pervasive Computing and Communications (PerCom).* 1–10.