Applying Prolog to Develop Distributed Systems

Nuno P. Lopes*, Juan A. Navarro†, Andrey Rybalchenko†, and Atul Singh‡
* *INESC-ID, Instituto Superior Técnico - Technical University of Lisbon*
† *Technische Universität München*
‡ *NEC Research Labs, Princeton, NJ*

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Development of distributed systems is a difficult task. Declarative programming techniques hold a promising potential for effectively supporting programmer in this challenge. While Datalog-based languages have been actively explored for programming distributed systems, Prolog received relatively little attention in this application area so far. In this paper, we investigate the applicability of a Prolog-based programming system, called DAHL, for the declarative development of distributed systems. For this task, we extend Prolog with an event-driven control mechanism and built-in networking procedures. Our experimental evaluation using a distributed hashtable data structure, a protocol for achieving Byzantine fault tolerance, and a distributed software model checker – all implemented in DAHL – indicates the viability of the approach.

1 Introduction

Declarative Networking is a promising direction in the quest for distributed programming systems that meets the challenges of building reliable and efficient distributed applications (Loo et al. 2005). As the name suggests, Declarative Networking advocates a high-level programming paradigm where the programmer specifies what has to be computed and communicated over the network, and then the compiler translates the specification into executable code. Its main applications are various network protocols, including sensor networks (Chu et al. 2007), fault tolerance protocols (Singh et al. 2008; Alvaro et al. 2009), distributed hash tables (Loo et al. 2005), and data replication systems (Belaramani et al. 2008).

Current implementations of Declarative Networking adapt Datalog for the domain of networking applications. The resulting programming languages have a bottom-up evaluation semantics where the evaluation of (Datalog) clauses causes the execution of corresponding networking actions. Since Datalog is a not a general purpose programming language, its adaptation for Declarative Networking required a reformulation of the language to allow some control over the flow of execution, to provide expressive data types, and to maintain a mutable state. Programmers often resort to C/C++ fragments on ordinary occasions (Singh et al. 2008), while research efforts are invested to better couple the required additional features with the Datalog evaluation model (Mao 2009; Alvaro et al. 2009).

In this paper we explore the applicability of Prolog as a basis for Declarative Networking. In contrast to Datalog, Prolog is a general purpose programming language. Since Prolog is considered to be a practical tool for programming in logic, its adaptation to distributed programming can focus only on the networked communication aspects. We put Prolog into an event-driven execution environment where messages received from the network are interpreted as (local) queries, and provide a collection of procedures for message passing-based communication. As a result, we obtain an extension of Prolog that can be applied for distributed programming. Its implementation, called DAHL[1], consists of a bytecode compiler and a runtime environment. DAHL builds upon an existing Prolog infrastructure (The Intelligent Systems Laboratory 2009) and a networking library (Mathewson and Provos 2009).

We evaluate DAHL on a range of distributed applications including the Chord distributed hash table (Stoica et al. 2001), the Byzantine fault tolerance protocol Zyzzyva (Kotla et al. 2007), and a distributed software model checker (Lopes and Rybalchenko 2010). DAHL implementations are comparable to existing Declarative Networking approaches in terms of succinctness, and do not require any C/C++ workarounds. Moreover, we also show that DAHL's performance is competitive with C++ runtimes produced with Mace, a tool that supports the development of robust distributed systems (Killian et al. 2007), while significantly reducing code size.

In this paper we present the following contributions:

- We demonstrate that Prolog is a suitable basis for the design of a programming language for Declarative Networking. Our approach exploits Prolog's strengths to provide general purpose programming features, while retaining its conceptual ties with the Declarative (Networking) paradigm.
- We provide an efficient and robust programming system for DAHL that includes a compiler and a runtime environment.
- We demonstrate the practicality of DAHL via an experimental evaluation on a range of distributed applications.

We organize the paper in the following way: In Section 2, we introduce DAHL using a simple spanning tree protocol as example. The programmer interface that allows the development of distributed applications is described in Section 3. We present implementation details of DAHL in Section 4, and evaluation results in Section 5. We also give a review of the related work in Section 6, and then conclude in Section 7.

2 DAHL by example

In this section, we illustrate DAHL by using an example program that implements a simple protocol for constructing a spanning tree overlay in a computer network. Tree-based overlay networks have received significant attention from the academic community (Castro et al. 2003; Chu et al. 2004; Jannotti et al. 2000; Banerjee et al. 2002) and have also seen successful commercial deployment (Li et al. 2007). In these

[1] Available at: `http://www7.in.tum.de/tools/dahl/`

```
:- event span_tree/2.

span_tree(Root, Parent) :-
  \+ tree(Root, _),
  assert(tree(Root, Parent)),
  this_node(ThisAddr),
  sendall(
    Node,
    neighbor(Node),
    span_tree(Root, ThisAddr)
  ).
```

Fig. 1. DAHL program to compute a spanning tree overlay.

tree overlays, after some network node has been selected to be the root node, we require that each other node is able to forward messages to the root node. After the spanning tree overlay is constructed, each node can send a message to the root by either using a direct link, if available, or relying on the ability of some neighbor to forward messages to the root. If a node is not connected to the root via a sequence of links then the node cannot send any messages to the root.

The overlay is constructed by propagating among the network nodes the information on how to forward to the root node. This information is given by the address of the next node towards the root. We assume that initially each node stores the addresses of its immediate neighbors in the (local) database. This information is loaded at startup by each node (e.g., at the command line or from a configuration file) into the `neighbor(`*Node*`)` table.

A node can directly access its neighbors by sending messages over the corresponding network links. At the initial step of the overlay construction, the designated root node, say *Root*, is triggered by sending it a message `span_tree(`*Root*, *Root*`)`. Then, the root node sends `span_tree(`*Root*, *Root*`)` to each neighbor node. At a neighbor, say *Node*, this message leads to the addition of the fact `tree(`*Root*, *Root*`)` to the database, thus, recording the possibility of reaching the tree root in a single step. Furthermore, *Node* propagates this information to its neighbors by sending a message `span_tree(`*Root*, *Node*`)`. Upon reception, each *Node*'s neighbor adds `tree(`*Root*, *Node*`)` to its database and continues the propagation.

Our implementation of the spanning tree protocol relies on a combination of Prolog with networking and distribution-specific extensions to achieve the goal, see Figure 1. When the initial message `span_tree(`*Root*, *Root*`)` arrives at *Root*, it is interpreted as a Prolog query. The query execution is carried out by the corresponding procedure `span_tree/2`, which is authorized to execute queries that arrive from the network due to the declaration `event span_tree/2`. The procedure `span_tree/2` uses standard Prolog predicates as well as our extensions. First, `span_tree/2` checks if the information how to reach the root node is already available. If it is the case, the execution of the procedure fails, and since the initiating query was issued by the network, DAHL ignores the failure and continues with the next message as soon as it arrives. Otherwise, a fact recording the root's reachability is added to the database. We propagate the corresponding information to

the neighbors, whose addresses are stored by each node as facts `neighbor/1` in the database. The message that is sent to each neighbor contains the sender address, which is required for the overlay construction. We obtain this address by using a DAHL built-in predicate `this_node/1`. The communication with the neighbors is implemented using a DAHL built-in procedure `sendall/3`, which is inspired by the "all solutions" predicates provided by Prolog, e.g., `findall/3` or `setof/3`. For each address that can be bound to `Node` by evaluating `neighbor(Node)`, the execution of `sendall` sends a message `span_tree(Root, ThisAddr)`, i.e., the message is sent to all neighbors.

In summary, our example shows that we can apply Prolog for developing distributed protocols by putting it into an event-driven execution environment and by extending the standard library with networking-specific built-in procedures. A more complex example is shown in Figure 6, which is an excerpt of our implementation of the Zyzzyva Byzantine fault tolerant protocol. In the rest of the paper, we briefly introduce the extensions and describe their interplay with Prolog for implementing a distributed hash table data structure, a protocol achieving the Byzantine fault tolerance, and a distributed version of a software model checking algorithm.

3 Programming interface for distributed applications

We now present the interface for developers to implement distributed applications. The interface consists of an event driven control and a set of primitives to send messages over the network. Our implementation of this interface is described later in Section 4.

Messages and event handlers Nodes communicate by exchanging messages represented by Prolog terms. When a message is received by a node, it triggers the evaluation of the matching event handler. An event handler corresponds to a Prolog predicate definition and its evaluation is done as a Prolog query.

The declaration

`:- event` *PredSpec*`, ..., ` *PredSpec*`.`

turns each predicate specified by *PredSpec* into an event handler for messages that match its specification. A predicate specification is an expression of the form `p/n` where `p` is a predicate name and `n` its arity. For example,

```
:- event q/2.

q(X, Y) :- Body.
```

declares the `q/2` predicate as the event handler for messages of the form `q(X, Y)`. In other words, the `event` declaration allows the evaluation of a predicate to be triggered when a matching message is received from the network.

In a running application, a node waits until a message is received from the network. When a message is received, and if the corresponding predicate is declared as an `event`, Prolog's standard evaluation strategy is used to compute the first solution to the message as if it was posed as a query. As the query is evaluated, the

event handler can modify the local state of the node, e.g., with `assert/retract`, or produce messages to send to other nodes. The solution to this query, or the failure to find a solution, is discarded, but the side effects of the evaluation are not. Messages that are not declared as events are also discarded. Event handlers triggered by different messages are evaluated atomically in sequence, i.e., the evaluation of a new message does not start until the evaluation of the previous one has finished. Atomic evaluation avoids concurrency issues that could arise when processing multiple messages at once.

DAHL provides the `send/2` and `sendall/3` built-in predicates to send messages over the network. The predicate

`send(`*Address*`, `*Message*`)`

sends *Message* to the node at *Address*. Evaluation of the predicate succeeds as soon as the underlying transport protocol reports the message as sent, and evaluation of the rest of the query continues. If an error occurs (e.g., *Address* is unreachable), the predicate fails and backtracks, e.g., to find an alternate destination. This is the default behavior and can be configured to throw exceptions or ignore errors instead.

Low level details, such as opening and closing network connections, are abstracted away and handled automatically by the DAHL runtime. If needed, developers can also access low level primitives to open/close connections themselves.

Multiple messages can be sent using

`sendall(`*Address*`, `*Generator*`, `*Message*`)`

which, for every solution of *Generator*, sends a *Message* to *Address*. A developer can use this predicate to broadcast a message to all neighbors of a node. For example,

```
sendall(N, neighbor(N), ping)
```

sends a `ping` message to every node `N` which is a solution to `neighbor(N)`. Moreover, both the *Address* and the *Message* of the `sendall` operation can be determined by the *Generator*. For example,

```
sendall(N, (task(T), assign(T, N)), solve(T)).
```

distributes a number of tasks among a set of nodes.

Low level implementations can optimize for particular usages of `sendall`. As an example, when *Message* does not depend on *Generator*, a network-level multicast/broadcast protocol can be used to provide a more efficient operation.

Another feature provided by the DAHL interface is that of alarms. Alarms are used by nodes to cause the evaluation of a local event handler at a specified time in the future. Similar to events, the declaration

`:- alarm` *PredSpec*`, ..., `*PredSpec*`.`

turns each predicate specified by *PredSpec* into an alarm handler. The predicate

`alarm(`*Message*`, `*MSecs*`)`

succeeds after setting up a reminder to insert *Message* in the local queue after *MSecs* have elapsed. The `alarm/2` predicate can also be used to trigger event handlers

declared with the `event` directive; but a predicate declared as `alarm` will never respond to messages from the network (e.g., produced with `send` or `sendall`).

Authentication When running a distributed protocol over an untrusted network, it is often required for messages to be signed in order to authenticate their origin. DAHL's interface allows to easily augment an application with authentication by replacing `send/2` with the predicate

`send_signed(`*Address*`,` *Message*`)`

that attaches authentication metadata to the *Message* sent to *Address*. Similarly, the `sendall_signed/3` predicate, analogous to `sendall/3`, is provided.

On the receiving end, the predicates

`signed_by(`*Address*`,` *Signature*`)`
`signed_by(`*Address*`)`
`signed`

check on demand whether the incoming message (and whose event handler is being evaluated) was properly signed. Additionally, if present, *Address* is unified with the address of the sender and *Signature* with the signature metadata. If the message was not signed, or had an invalid signature, these predicates fail.

Since cryptographic operations are often computing intensive, these predicates allow the programmer to schedule the validation of signatures at an appropriate time in the evaluation of an event handler. For example,

```
request(Req) :- valid(Req), signed_by(Addr), ...
```

checks the validity of a request before performing a, possibly more expensive, validation of the signature. This strategy is applied in the definition of `request/1` in our implementation of Zyzzyva (Figure 6).

Authenticity in DAHL is based on OpenSSL's implementation of HMAC for signing messages and MD5 for computing message digests. Alternative crypto-algorithms can be selected and accessed through the same high-level interface.

4 Implementation

The software architecture of DAHL is shown in Figure 2. It consists of a DAHL compiler (based on SICStus Prolog compiler from The Intelligent Systems Laboratory 2009), a high-performance event dispatching library (libevent from Mathewson and Provos 2009), the OpenSSL library to provide the cryptographic primitives in the language, the DAHL runtime, and DAHL applications. We use the off-the-shelf SICStus Prolog compiler to quickly build the DAHL system and utilize its industrial-strength performance and robustness for achieving high performance. We do not describe the details of how we interfaced libevent and OpenSSL since they are standard, instead we describe in detail the novel aspects of DAHL: how the runtime works, some optimizations that were implemented, and the networking aspects.

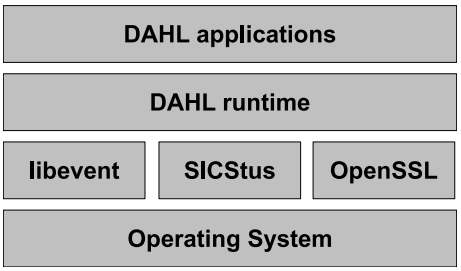

Fig. 2. The DAHL software stack.

Runtime DAHL's runtime consists of a library written in Prolog (with around 460 lines of code), which implements the built-in predicates, and a networking back-end written in C (around 450 lines). It is the networking back-end that interfaces with both libevent and OpenSSL. This back-end interfaces with Prolog through stubs generated automatically by the SICStus Prolog compiler.

DAHL programs are interpreted directly by the SICStus Prolog compiler, but under the DAHL runtime control. The main program in execution is a loop that is part of the runtime, and a DAHL program's code is only called when an appropriate event arrives from the network, or when a timer is triggered. Those events are processed by libevent.

Figure 3 shows the execution flow for processing a message that arrives from the network (steps 1–4), and for a message that is sent from an application (steps 5–6) in more detail. When a message arrives from the network, the operating system dispatches it to libevent (step 1), which queues the message. Then, when the DAHL runtime asks for the next message, libevent picks one arbitrarily and delivers it to the DAHL network back-end (step 2). The DAHL network back-end then deserializes the message and calls the runtime dispatcher (in Prolog) through a stub (step 3). Finally, the dispatcher calls the corresponding event handler of the application (step 4). When a DAHL application sends a message, the message is first handed over to the DAHL runtime through a stub (step 5). The runtime then serializes the message and delegates the network transmission to the operating system (step 6).

Optimizations We implemented several optimizations in the DAHL runtime to improve its performance. Here, we present these optimizations in detail. The deserialization of network messages was a CPU-intensive operation since the SICStus Prolog compiler implements this operation in Prolog through a complex process chain. Since each message sent was serialized to a single atom, it led to an explosion in memory usage because the SICStus Prolog compiler aggressively caches all atoms. We therefore implemented our own custom deserialization in C to improve the performance. This resulted in a performance improvement of the deserialization function of about 70%.

As described before, the main loop is implemented in Prolog, and it calls a function in C that "produces" events through libevent, which are then dispatched from within the Prolog environment. After an event is dispatched and processed, the

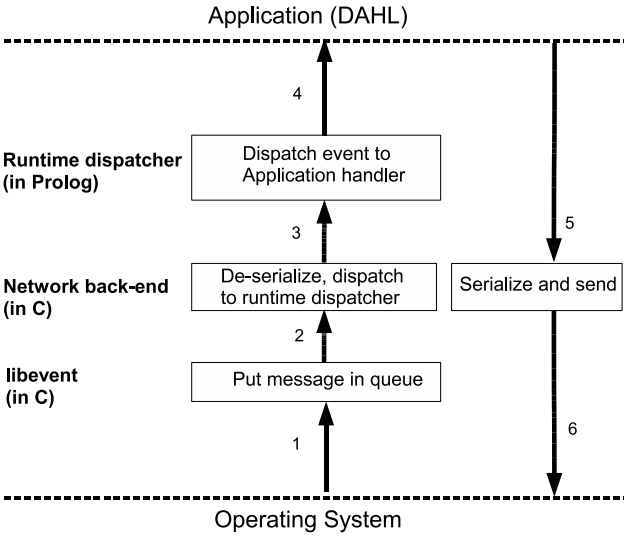

Fig. 3. Internals of DAHL runtime shown by tracing the flow of message processing.

loop backtracks until the beginning of it. This provides an important advantage, which is that every event/alarm handler is executed in a "clean" environment, as all the garbage possibly left by a previous handler is discarded. Moreover, it improves the performance of the garbage collector (GC), as the SICStus Prolog compiler will delete most of such garbage when backtracking as an optimization, reducing the overhead of the GC. Our tests show that without this environment cleanup, the overhead of the GC would be noticeable (from 8% to 45%).

Network Support Currently all the network messages are sent using the TCP protocol, which requires establishing a connection before the first contact. The DAHL runtime automatically establishes these connections when needed, and caches them indefinitely for future contacts. It is straightforward to replace TCP with UDP, though the application needs to have mechanisms to handle message loss.

5 Evaluation

In this section, we present an evaluation of DAHL in terms of run-time performance, language expressiveness, and succinctness of programs. Implementations of networking protocols, like Chord (Stoica et al. 2001) and Zyzzyva (Kotla et al. 2007), as well as CPU-bound applications like D'ARMC (Lopes and Rybalchenko 2010) demonstrate the applicability of DAHL in the development of real-life and complex systems. We compare the results with alternative implementations of these protocols in P2 (Loo et al. 2006), the original implementation of Declarative Networking, and Mace (Killian et al. 2007), an extension of C++ with networking capabilities and a state-machine specification language.

Table 1. *Comparing raw network performance of P2, DAHL, Mace, Mace compiled with '-O2' optimizations, and plain C as the maximum number of pings responded by the server in a second.*

P2	DAHL	Mace	Mace (w/ -O2)	C
230	14,000	14,221	21,937	142,800

5.1 Raw Performance

To evaluate the performance of the DAHL runtime, we performed a test to compare the performance of P2, Mace, C, and DAHL. We performed a simple network ping-pong experiment. One of the machines (called a client) sends a small 20 Byte 'ping' message to the other machine (called a server) which immediately responds with a small 20 Byte 'pong' message. We used as many client machines as needed to saturate the server in order to measure its raw throughput. The measurement of the number of requests served per second was done at the server. The machines were connected by a gigabit switch with a round trip latency of 0.09 ms, and both the network and the machines were unloaded. The results are presented in Table 1.

First, we note that the DAHL runtime outperforms P2's performance. We believe that the reason behind P2's poor performance is that the runtime of P2 is not yet optimized while DAHL uses the SICStus 4 compiler that has been already optimized. Second, DAHL is as fast as Mace. However, given that Mace is a restricted form of C++, it can exploit powerful C++ optimizing compilers. For example, with the '-O2' set of optimizations of gcc 4.1, Mace's performance improves by 60%. As an upper bound on the performance, we also present the performance of a C implementation and note that all the systems that strive to improve the analysis capability—by providing higher level programming abstractions which are also more amenable to static analysis and program verification techniques—are an order of magnitude slower.

5.2 Chord

In this subsection, we evaluate the performance of an implementation of Chord (a distributed hash table) in DAHL. Our implementation of Chord implements all features detailed in the original paper (Stoica et al. 2001). To compare with the P2 Chord implementation, we obtained the latest release of P2.[2] Unfortunately, we were unable to get P2 Chord running in our local setup. We therefore cite results from their paper (Loo et al. 2005).

Setup We used ModelNet (Vahdat et al. 2002) to emulate a GT-ITM transit-stub topology consisting of 100 to 500 stubs and ten transit nodes. The stub-transit links

[2] Version 3570 in `https://svn.declarativity.net/p2/trunk`.

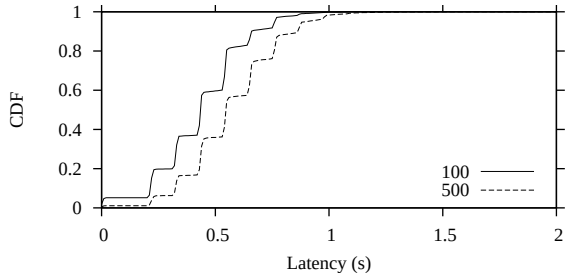

Fig. 4. Chord: Lookup latency distribution.

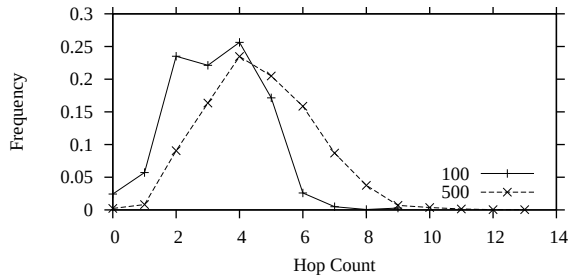

Fig. 5. Chord: Hop count distribution.

had a latency of 2 ms and 10 Mbps of bandwidth, while transit-transit links had a latency of 50 ms and 100 Mbps of bandwidth. We used 10 physical hosts, each with dual core AMD Opteron 2.6 Ghz processor with 8GB of RAM, and running Linux kernel version 2.6.24. We ran 10 to 50 virtual nodes on each physical node, producing a population of 100 to 500 nodes. In each experiment, neither the CPU nor the RAM were the bottleneck. This setup reproduces the topology used by the P2 experiments in (Loo et al. 2005), although they used Emulab (White et al. 2002).

Static Membership Our first goal was to see if the DAHL implementation met the high-level properties of Chord. We have first evaluated our implementation by performing 10,000 DHT 'lookup' requests generated from random nodes in the network for a random set of keys. The lookups were generated after waiting for five minutes after the last node joined in order to let the network stabilize.

In Figure 4, we present the cumulative distribution of latency incurred to receive the response to the lookup requests with 100 and 500 nodes. The results are comparable or better than the published results for P2 Chord (Loo et al. 2005).

In Figure 5, we present the frequency distribution of the number of hops taken to complete the lookups. As expected, the maximum number of hops taken is under the theoretical limit of $\lceil \log N \rceil$.

Dynamic Membership Our implementation of Chord in DAHL also handles churn. In this experiment, we used 500 nodes, each one maintaining four successors and performing finger fixing every 10 seconds and successor stabilization every 5 seconds.

This configuration is similar to the setup of P2 Chord. We generated artificial churn in our experiment by killing and joining nodes at random with different session times by following the methodology presented in (Rhea et al. 2004).

We obtained lookup consistency of 96% for average session times of 60 minutes, which is comparable with other implementations of Chord.

Summary Our results show that our implementation of Chord in DAHL covers the major algorithmic aspects of the protocol and that its run-time performance is competitive with P2 Chord.

5.3 Zyzzyva

In this subsection, we evaluate the implementation of Zyzzyva in DAHL. For reference, and to give a flavor of the code written in DAHL, we include a fragment of the implementation of its first phase in Figure 6. We present the peak throughput for the normal case, and the throughput after killing a backup replica. The goal of our experiments is to show that our implementation covers a significant part of Zyzzyva protocol and to show that its performance is reasonable. We compare the performance of DAHL Zyzzyva with the publicly available C++ implementation of Zyzzyva (Kotla et al. 2008).

Setup We use four physical machines as servers to tolerate one Byzantine faulty server and vary the number of clients to measure the peak throughput. Both the server and client machines have identical characteristics as previous experiment. The clients send requests with 0 B payload, the execution cost of each operation at the servers is zero, and we measure the peak throughput sustained by the servers.

Implementation We use OpenSSL's HMAC+MD5 cryptographic hash function in DAHL to perform critical digest and signing operations. Our implementation uses TCP as the transport protocol, we do not yet use network broadcast feature, and do not implement batching. Our implementation takes checkpoint at the rate of every 128 requests, which is standard in existing implementations. We do not implement state transfer mechanism to bring the slow replicas up-to-date.

First case performance In this experiment, we present the peak throughput of Zyzzyva without failures where requests are completed in single phase. This result serves to measure the baseline functionality of Zyzzyva. The results are presented in Table 2. We observe that the performance of DAHL's Zyzzyva is ∼10 times slower than the C++ implementation. However, as Clement et al. (2009) observe, the penalty of using DAHL over C++ will diminish as the application level overhead starts to dominate. For example, with an application that consumes approximately 100 μs per operation, Zyzzyva will deliver throughput of 9 Kreq/s while the implementation in DAHL will deliver approximately 3 Kreq/s, bringing down the penalty to 3X.

```
 1: :- dynamic seqno/1, pending/3, cache/4.      | Program state, and
 2: :- event request/1, process/2.               | event declarations.
 3:
 4: request(Req) :-                              | When I get a request ...
 5:   this_node(ThisAddr),                       | Find my own address,
 6:   primary(ThisAddr),                         | if I'm the primary,
 7:   signed_by(Src),                            | and got a signed request,
 8:   \+ pending(_, Src, Req),                   | which I haven't seen before,
 9:   count(pending(_, _, _), N),†               | count the previous requests,
10:   Id is N + 1,                               | to produce a new id, and
11:   assert(pending(Id, Src, Req)),             | add the new request as pending.
12:   batch_size(Size),                          | Recall the size of a batch,
13:   Id =:= Size,                               | if there are enough requests,
14:   start_new_batch.                           | start the protocol for this batch.
15:
16: start_new_batch :-                           | When starting a new batch ...
17:   findall(
18:     (Id, Src, Req),                         | Collect all the pending requests,
19:     retract(pending(Id, Src, Req)),         | removing them from the store,
20:     Batch                                   | and group them in a batch.
21:   ),
22:   retract(seqno(Seq)),                      | Get the next sequence number.
23:   sendall_signed(
24:     Node,                                   | Ask all nodes,
25:     replica(Node),                          | that happen to be replicas,
26:     process(Batch, Seq)                     | to process this batch.
27:   ),
28:   Next is Seq + 1,                          | Increment the sequence no.,
29:   assert(seqno(Next)).                      | and store the new value.
30:
31: process(Batch, Seq) :-                       | When processing a batch ...
32:   primary(Primary),                         | Look up who is the primary,
33:   signed_by(Primary),                       | as this should be the one asking.
34:   findall(_, (
35:     member((Id, Src, Req), Batch),          | Unpack the batch,
36:     process_req(Seq-Id, Src, Req)           | and process each request.
37:   ), _).
38:
39: process_req(Seq, Src, Req) :-                | When processing a request ...
40:   ( cache(Seq, Src, Req, Out) ->            | If I've seen this request before
41:     send_signed(Src, reply(Seq, Req, Out))  | reply with the cached response.
42:   ;                                         | Otherwise,
43:     \+ cache(Seq, _, _, _)),                | if it's a new sequence no.,
44:     compute_output(Req, Out),               | compute the output,
45:     assert(cache(Seq, Src, Req, Out)),      | store it on the cache,
46:     send_signed(Src, reply(Seq, Req, Out))  | and send it back to the client.
47:   ).
```

Fig. 6. Initial phase of Zyzzyva with batching optimization.

†`count/2` is a non-standard Prolog extension that counts the number of solutions of a given goal.

Table 2. *Zyzzyva: single phase and two phase performance for 0-byte payload.*

	DAHL Zyzzyva	C++ Zyzzyva
Single phase	4.5K	40K
Second phase	2.5K	20K

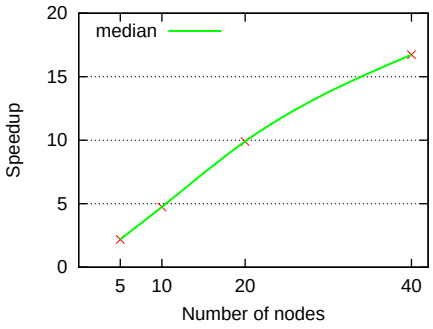

Fig. 7. Median speedup of D'ARMC with varying number of nodes.

Second phase performance In this experiment, we present the peak throughput of Zyzzyva when upto F replicas are faulty and prevent requests from completing in the single phase. This requires client to initiate the second phase, requiring more computation and network resources at the replicas, resulting in lower performance compared to the previous result based on single phase. Again, our results show that DAHL implementation is slower compared to its counterpart in C++ owing to a slower runtime.

Summary The primary goal of our evaluation was to check if our implementation is comprehensive and faithful, and to evaluate its performance. Our results show that the current implementation covers a significant portion of the protocol features but the performance is lower compared to C++ implementation.

5.4 D'ARMC

D'ARMC (Lopes and Rybalchenko 2010) is a distributed software model checker that was implemented in DAHL. D'ARMC is a CPU-bound application, and therefore shows that DAHL can be used to implement more applications than mere network protocols. The median speedup achieved by D'ARMC in a set of benchmarks is shown in Figure 7. The benchmarks consist in a set of automata-theoretic models from the transportation domain and a standard hybrid system example.

As can be seen in Figure 7, D'ARMC shows a linear speedup with a varying number of machines, and the efficiency is about 50%. A more extensive evaluation can be found in (Lopes and Rybalchenko 2010).

5.5 Code Size

Our implementations of both Chord and Zyzzyva are comparable in size to the P2 implementations in terms of lines of code (LoC). For example, DAHL Chord is implemented in 215 LoC while the P2 Chord is implemented in 211 LoC. These sizes are an order of magnitude more succinct compared to a C/C++ implementation.

6 Related work

In the previous section we have already compared DAHL with two other related systems that help the programmer to build distributed applications, P2 (Loo et al. 2006) and Mace (Killian et al. 2007). Both languages have been successfully used for the implementation of important networked systems and protocols, and serve as a research platform for the development of specialized variants—see Declarative Languages and Systems (2009) for further pointers—as well as verification tools (Killian et al. 2007; Yabandeh et al. 2009; Navarro and Rybalchenko 2009; Wang et al. 2009; Navarro et al. 2009).

Alternative approaches that attempt to extend Datalog for use in a distributed environment, while trying to overcome the pitfalls of early Declarative Networking implementations, are Meld (Ashley-Rollman et al. 2007; Ashley-Rollman et al. 2009), WIND (Mao 2009) and Netlog (Grumbach and Wang 2010). A common feature of these projects is that they all argue that a 'pure' Datalog based language is not appropriate for the development of stateful applications. The authors of Meld show that a limited declarative language can be used to program behavior in ensembles; the authors of WIND propose the use of syntactic 'salt' to discourage, but still allow, the use of imperative features; while the authors of Netlog augment Datalog rules with annotations to explicitly control whether tuples are stored or sent over the network.

In the broader picture of designing high-level languages for concurrent and distributed programming, a prime example is Erlang (Armstrong et al. 1993). Erlang is based on the functional programming paradigm and, similar to our approach, incorporates distribution via explicit message passing between processes. A related approach suggests using the Lua programming language to implement distributed systems (Leonini et al. 2009).

Some projects also aim to exploit the use of functional programming languages at lower layers of the network protocol deign. Foxnet, for example, implements the standard TCP/IP networking protocol stack in ML (Biagioni et al. 2001); while Melange provides a language to describe Internet packet protocols, and generates fast code to parse/create these packets (Madhavapeddy et al. 2007). Similarly, the KL1 logic based language has been used to model and exploit physical parallelism in the PIM operating system (Bal 1993).

Previous work has also explored the use of Prolog to deal with concurrency and parallelism, a comprehensive review is given by Gupta et al. (2001). Most of this work, however, deals with the problem of using Prolog to paralellize an otherwise sequential task. Recent advances in this direction are discussed by (Casas

et al. 2008). Our work explores, instead, the use of Prolog as a general purpose programming language to implement distributed applications.

7 Conclusion

From our experience with applying Prolog for distributed programming we draw the following conclusions.

In combination with event-driven control and networked communication primitives, Prolog offers a programming language that is sufficiently expressive and well-suited for the implementation of distributed protocols. In our experiments, we did not rely on any C/C++ extensions as there was no need to compensate absence of certain programming constructs, as it is common for the P2 system for declarative networking that is Datalog-based. Instead, we used the data type, control structures, and the database facility provided by Prolog. By using Prolog as a basis we avoided any major compiler/runtime/libraries implementation efforts, which often become an obstacle when implementing a new programming language. By not starting from scratch and relying on the existing Prolog infrastructure, we obtain correctness and efficiency of program execution out of the box.

References

Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., Hellerstein, J. M., and Sears, R. C. 2009. Boom: Data-centric programming in the datacenter. Tech. Rep. UCB/EECS-2009-98, EECS Department, University of California, Berkeley. Jul.

Alvaro, P., Condie, T., Conway, N., Hellerstein, J. M., and Sears, R. 2009. I Do Declare: Consensus in a Logic Language. In *Workshop on Networking Meets Databases (NetDB)*. Big Sky, Montana, USA.

Armstrong, J., Virding, R., Wikstrm, C., and Williams, M. 1993. *Concurrent Programming in ERLANG*. Prentice Hall, Englewood Cliffs, New Jersey.

Ashley-Rollman, M. P., Goldstein, S. C., Lee, P., Mowry, T., and Pillai, P. 2007. Meld: A declarative approach to programming ensembles. In *Proc. of IEEE/RSJ Conference on Intelligent Robots and Systems (IROS)*.

Ashley-Rollman, M. P., Lee, P., Goldstein, S. C., Pillai, P., and Campbell, J. D. 2009. A language for large ensembles of independently executing nodes. In *Proc. of the 25th International Conference on Logic Programming (ICLP)*.

Bal, H. E. 1993. Evaluation of kl1 and the inference machine. *Future Gener. Comput. Syst. 9,* 2, 119–125.

Banerjee, S., Bhattacharjee, B., and Kommareddy, C. 2002. Scalable Application Layer Multicast. In *ACM Special Interest Group on Data Communication (SIGCOMM02)*. Pittsburg, PA, USA.

Belaramani, N., Zheng, J., Nayate, A., Soulé, R., Dahlin, M., and Grimm, R. 2008. Padre: A policy architecture for data replication systems. In *UT Austin Technical Report*. Texas, USA.

Biagioni, E., Harper, R., and Lee, P. 2001. A network protocol stack in standard ml. *Higher Order Symbol. Comput. 14,* 4, 309–356.

Casas, A., Carro, M., and Hermenegildo, M. 2008. A high-level implementation of non-deterministic, unrestricted, independent and-parallelism. In *ICLP: Logic Programming*. LNCS, vol. 5366. Springer, 651–666.

Castro, M., Druschel, P., Kermarrec, A., Nandi, A., Rowstron, A., and Singh, A. 2003. Splitstream: High-bandwidth multicast in a cooperative environment. In *SOSP*. Bolton Landing, NY, USA.

Chu, D., Popa, L., Tavakoli, A., Hellerstein, J. M., Levis, P., Shenker, S., and Stoica, I. 2007. The design and implementation of a declarative sensor network system. In *SenSys*. ACM.

Chu, Y., Ganjam, A., Ng, T., Rao, S., Sripanidkulchai, K., Zhan, J., and Zhang, H. 2004. Early Experience with an Internet Broadcast System Based on Overlay Multicast. In *USENIX Annual Technical Conference (USENIX04)*. Boston, MA, USA.

Clement, A., Marchetti, M., Wong, E., Alvisi, L., and Dahlin, M. 2009. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*.

Declarative Languages and Systems. 2009. `http://declarativity.net`.

Grumbach, S. and Wang, F. 2010. Netlog, a rule-based language for distributed programming. In *PADL'10: Proceedings of the Eleventh International Symposium on Practical Aspects of Declarative Languages*. Number 5937 in Lecture Notes in Computer Science. Springer, 88–103.

Gupta, G., Pontelli, E., Ali, K., Carlsson, M., and Hermenegildo, M. 2001. Parallel execution of prolog programs: a survey. *ACM Transactions on Programming Languages and Systems (TOPLAS) 23,* 4, 472–602.

Jannotti, J., Gifford, D., Johnson, K. L., Kaashoek, M. F., and Jr, J. W. O. 2000. Overcast: Reliable Multicasting with an Overlay Network. In *OSDI*. San Diego, CA, USA.

Killian, C., Anderson, J. W., Braud, R., Jhala, R., and Vahadat, A. 2007. Mace: Language support for building distributed systems. In *PLDI*.

Killian, C. E., Anderson, J. W., Jhala, R., and Vahdat, A. 2007. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*. USENIX.

Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. 2007. Zyzzyva: speculative byzantine fault tolerance. *ACM SIGOPS Operating Systems Review 41,* 6, 45–58.

Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. 2008. `http://cs.utexas.edu/~kotla/RESEARCH/CODE/ZYZZYVA/`.

Leonini, L., Rivière, É., and Felber, P. 2009. Splay: Distributed systems evaluation made simple. *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, 185–198.

Li, B., Xie, S., Keung, G. Y., Liu, J., Stoica, I., Zhang, H., and Zhang, X. 2007. An empirical study of the coolstreaming+ system. *IEEE Journal on Selected Areas in Communication (JSAC) 25,* 9, 1627–1639.

Loo, B. T., Condie, T., Garofalakis, M., Gay, D. E., Hellerstein, J. M., Maniatis, P., Ramakrishnan, R., Roscoe, T., and Stoica, I. 2006. Declarative networking: Language, execution and optimization. In *SIGMOD*. ACM.

Loo, B. T., Condie, T., Hellerstein, J. M., Maniatis, P., Roscoe, T., and Stoica, I. 2005. Implementing declarative overlays. In *SOSP*.

Lopes, N. P. and Rybalchenko, A. 2010. Distributed and predictable software model checking. In *(submitted)*.

Madhavapeddy, A., Ho, A., Deegan, T., Scott, D., and Sohan, R. 2007. Melange: creating a "functional" internet. In *EuroSys '07*. ACM, New York, NY, USA, 101–114.

Mao, Y. 2009. On the declarativity of declarative networking. In *Proceedings of the Workshop on Networking Meets Databases, NetDB 2009*.

Mathewson, N. and Provos, N. 2009. *libevent Documentation*. Release 1.4.9.

Navarro, J. A. and Rybalchenko, A. 2009. Operational semantics for declarative networking. In *PADL*.

Navarro, J. A., Rybalchenko, A., and Singh, A. 2009. Cardinality abstraction for declarative networking. In *CAV: Camputer Aided Verification*.

Rhea, S., Geels, D., Roscoe, T., and Kubiatowicz, J. 2004. Handling Churn in a DHT. In *ATEC*. USENIX.

Singh, A., Das, T., Maniatis, P., Druschel, P., and Roscoe, T. 2008. BFT protocols under fire. In *NSDI*.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 2001. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM*.

The Intelligent Systems Laboratory. 2009. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science. Release 4.0.5.

Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostić, D., Chase, J., and Becker, D. 2002. Scalability and Accuracy in a Large-Scale Network Emulator. In *OSDI*.

Wang, A., Basu, P., Loo, B. T., and Sokolsky, O. 2009. Declarative network verification. In *PADL*.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. 2002. An integrated experimental environment for distributed systems and networks. In *OSDI*.

Yabandeh, M., Knezevic, N., Kostic, D., and Kuncak, V. 2009. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *NSDI*.