Netlog, a Rule-Based Language for Distributed Programming

Stéphane Grumbach[1] and Fang Wang[2]

[1] INRIA-LIAMA, PO Box 2728, Beijing 100190, P.R. China
Stephane.Grumbach@inria.fr
[2] GUCAS & LCS, ISCAS, PO BOX 8718, Beijing 100080, P.R. China
wangf@ios.ac.cn

Abstract. We propose a rule-based language, Netlog, to express distributed applications such as communication protocols or P2P applications in a declarative manner. The language extends Datalog with communication primitives, as well as aggregation and non-deterministic constructs, standard in network applications. Our contribution is twofold. First we define a sound distributed fixpoint semantics, which takes explicitly into account the in-node behavior as well as the communication between nodes, and solves semantic problems raised in declarative networking. Second, we show that syntactic restrictions over the programs can ensure polynomial bounds on the complexity (time and message) of the distributed execution. The language has been implemented and runs over a virtual machine, Netquest, which relies on a DBMS. Netlog programs are partly compiled into SQL queries, which makes them portable over heterogeneous architecture.

1 Introduction

The trend towards ubiquitous environment is accelerated with wireless technologies interconnecting an increasing number of heterogeneous devices. Their intermittent availability, the dynamicity of the networks, as well as the data intensive applications envisioned raise considerable challenges. One of the fundamental barriers today to their development is the **lack of programming abstraction** [15].

The declarative networking approach, initially proposed in [12] has been shown to offer a nice paradigm to express in a declarative manner network applications. Nevertheless, as shown in particular in [16], its semantics has not been formally defined and suffers from severe ambiguities. In this paper, we propose a new rule-based language that (i) integrates a collection of rich primitives required in networking applications, (ii) admits a well-defined distributed fixpoint semantics, (iii) has been implemented on the Netquest system, and tested over simulated networks, and finally (iv) supports optimization and allows to bound the complexity of the distributed execution.

Smart devices are usually dedicated systems based on ad hoc models, which are not generic enough to support the needs of future applications (flexibility, scalability, ease to produce and maintain, etc.). The deployment of a sensor network for instance is a tedious task which requires an expertise in the underlying OS and hardware.

The **separation of a logical level**, accessible to users and applications, **from the physical layers** constitutes the basic principle of Database Management Systems. It is at the origin of their technological and commercial success [17]. This fundamental contribution of Codd in the design of the relational model of data, has lead to the development of universal high level query languages, that all vendors recognize, as well as to query processing techniques that optimize the declarative queries into (close to) optimal execution plans.

Declarative query languages have already been used in the context of networks. Several systems for sensor networks, such as TinyDB [14] or Cougar [7] offer the possibility to write queries in SQL. These systems provide solutions to perform energy-efficient data dissemination and query processing. A distributed query execution plan is computed in a centralized manner with a full knowledge of the network topology and the capacity of the constraint nodes, which optimizes the placement of subqueries in the network [19]. Declarative methods have been used also for unreliable data cleaning based on spatial and temporal characteristics of sensor data [9] for instance.

Another application of the declarative approach has been pursued at the network layer. The use of recursive query languages has been initially proposed to express communication network algorithms such as routing protocols [13] and declarative overlays [12]. This approach, known as **declarative networking** is extremely promising. It has been further pursued in [11], where execution techniques for Datalog are proposed. Distributed query languages thus provide new means to express complex network problems such as node discovery [3], route finding, path maintenance with quality of service [6], topology discovery, including physical topology [5], secure networking [1], or adaptive MANET routing [10].

The problems of semantics raised by declarative networking, motivated us to introduce a new language. As NDlog [11] for instance, it relies on the deductive languages [18] developed in the 80's in the field of databases, but with important differences that facilitate both execution and semantics. One of the fundamental characteristics of the proposed language is that **Netlog programs are local**. One node cannot access the memory of another node neither for write nor for read instructions. This simplifies greatly the semantics of negation. It also facilitates the design of secure protocols. Netlog also extends classical recursive rule languages with arithmetic, aggregate functions, as well as a non-deterministic choice, which are required for many distributed problems, such as those involved in networking protocols.

Fixpoint logics and rule-based languages have been widely studied in the classical centralized setting [2]. The main originality of the **distributed fixpoint semantics** proposed in this paper is to take explicitly into account the communication between nodes, and in particular the routing issue. Programs can generate messages to arbitrary destinations, which have to be routed to some neighbor following a routing strategy.

The distributed fixpoint semantics is defined for asynchronous systems. On each node, a local round consists of a computation phase followed by a communication phase. During the computation phase, the program updates the local data and produces messages to send. During the communication phase, the router transmits the incoming messages to the program, and routes the outgoing messages. In the present setting, a message can be routed if a route is found in the data on the node, otherwise it is discarded. Other choices can of course equally be made.

It has been widely shown now that rule-based languages allow to obtain code about two orders of magnitude more concise than standard imperative programming languages. But at this stage, this code is not necessarily simple to write. The main challenge for declarative networking is to develop techniques for rewriting programs which are simple to write into equivalent programs, which admits efficient execution. This means optimizing the programs, making them adapt to their context, with different execution schemes.

In this paper, we concentrate on the complexity, and show that syntactic restriction on the rules can enforce complexity bounds on their execution. We consider three complexity measures, the distributed time and the message complexity, which are classical in distributed computing, and the in-node complexity, which is interesting for restricted terminals. We show that a restricted class of programs, namely the **well-behaved programs**, admit **polynomial complexity bounds** for these three measures.

We have developed a virtual machine, Netquest, which runs Netlog programs according to the distributed fixpoint semantics. It relies on an embedded DBMS, which stores the data as well as the programs on the nodes of the network. The Netlog programs are essentially compiled into SQL queries, which are then executed by the DBMS. The Engine manages the iteration of the queries. This choice of implementation was motivated by the fact that an increasing number of devices now support embedded DBMS's. It simplifies the development, makes the Netquest system easily portable over heterogeneous devices and networks, and supports data intensive applications.

We have used Netlog to program a large set of problems from classical distributed algorithms to networking, from sensor networks to P2P games. They confirmed the conciseness of the code, validated our semantics, as well as the expected behavior derived from the syntactic form of the rules.

The paper is organized as follows. In the next section, we present the computation model. The Netlog language is presented through examples in Section 3. Section 4 is devoted to the distributed fixpoint semantics. In Section 5, we study the complexity of Netlog programs. A brief presentation of the implementation is done in Section 6.

2 The Computation Model

We next introduce the computational model on which the Netlog rules are executed. We consider a message passing model for distributed computation

[4], based on a communication network whose topology is given by a graph $\mathcal{G} = (V_\mathcal{G}, Link)$, where $V_\mathcal{G}$ is the set of nodes, and *Link* denotes the set of bidirectional *communication links* between nodes. The nodes have a unique *identifier*, *Id*, taken from $1, 2, \cdots, n$, where n is the number of nodes. Each node has distinct local ports for distinct links incident to it. The control is fully distributed in the network, and there is no shared memory.

The *communication* between nodes rely on messages which have the following format: $message :=< content, destination >$. We thus distinguish between two parts in each message: (i) the content of the message, and (ii) its destination. The *content* is restricted to facts derived by the Netlog rules. The *destination* is either a node *Id*; or *nil* (the message is sent to neighbor nodes); or *all* (the message is broadcasted to all nodes).

We distinguish between *computation events*, performed in a node, and *communication events*, performed by nodes which cast their messages to their neighbors. On one node, a *computation phase* followed by a *communication phase* is called a *local round* of the distributed computation.

All the nodes have the same architecture and the same behavior. We make in general no particular assumption on the distributed system, which might be asynchronous, have failure, and rely on moving nodes. The **architecture** of each node is composed of three main components, (i) a router, handling the communication with the network; (ii) an engine, executing the Netlog programs; and (iii) a local data store to maintain the information (data and programs) local to the node.

The modules of the system on a node α at local round ℓ behave as follows:

- **Router.** During the computation phase, the router queues the incoming messages on the *reception queue*, $\mathcal{R}^\alpha(\ell)$, and the messages to push produced by the Engine on the *emission queue*, $\mathcal{P}^\alpha(\ell)$.

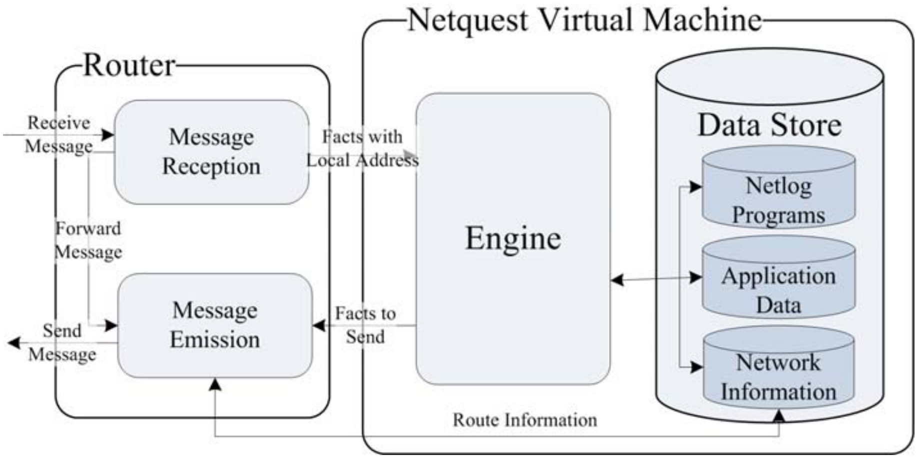

Fig. 1. The node architecture

When the communication phase starts, the messages on the reception queue, $\mathcal{R}^\alpha(\ell)$, are sorted according to their destination. (i) If their destination is α (the node *Id*), *nil*, or *all*, their content, grouped in $\mathcal{L}^\alpha(\ell)$, is transfered to the Engine. (ii) If their destination is another node *Id*, or *all*, the messages, grouped in $\mathcal{F}^\alpha(\ell)$, are put on the emission queue, $\mathcal{P}^\alpha(\ell)$. The reception queue is then emptied.

Then, each message on the emission queue, $\mathcal{P}^\alpha(\ell)$, is handled. Either its destination is *nil* or *all*, and the message is sent to all neighbors. Otherwise, a route to the desired destination is queried in the *Route* relation in the data store. The message is sent to the next hop on that route if it is found, and otherwise discarded[1].

- **Deductive Engine.** It processes the programs during the computation phase. First, the programs, that can be activated by the new facts in $\mathcal{L}^\alpha(\ell)$, are loaded. The rules are then run till no rules can be executed to derive new facts, new derived facts (in $\mathcal{I}^\alpha(\ell+1)$) are stored in the data store, and messages produced are pushed to the emission module, $\mathcal{P}^\alpha(\ell)$, of the router. The behavior of the Engine follows the semantics of the language presented in the sequel.
- **Local Datastore.** It handles two sorts of information: all the data of the node, whether related to networking issues (e.g. network topology, routes, bandwidth, etc.) or applications, as well as the rules of the protocols.

The Datastore contains all data, which are all modeled as relations. Some predefined relations are used by the system. It is the case of the two relations *Link* of arity 2 and *Route* of arity 3:

$$\begin{aligned} \mathit{Link} &= \quad (\mathit{Source}, \mathit{Destination}) : && \mathbb{N} \times \mathbb{N} \\ \mathit{Route} &= (\mathit{Source}, \mathit{Nexthop}, \mathit{Destination}) : && \mathbb{N} \times \mathbb{N} \times \mathbb{N} \end{aligned}$$

The relation *Link* is read-only. It is maintained by the underlying network monitoring. Each node has the fragment of the relation *Link* with its neighbors. The relation *Route* on the other hand is computed by programs, and is used by the Emission module of the Router. It therefore plays a particular role in the semantics of Netlog programs. If no routes are available, the communication is restricted to neighbors. Note that in some examples, we use relations of larger arity for links and routes with their costs for instance. The two built-in relations *Link* and *Route*, are then defined as views over more complex links and routes.

A Netlog program starts with declarations which include the data formats (relations) used, as well as some initial facts to store in the local store. They are installed on the data stores when programs are loaded.

3 The Netlog Language

We introduce the language and its primitives through the fundamental example of route computation. Netlog relies on recursive rules, of the form $\mathit{head} \ :- \ \mathit{body}$,

[1] Other strategies can be implemented, such as search for a route, forward to other nodes, or failure messages.

which informally mean that if the body is true then the head can be derived. Let us recall the recursive rules which define the transitive closure of *Link* in a centralized environment:

$$TC(x, y) :- Link(x, y). \tag{1}$$

$$TC(x, z) :- Link(x, y); TC(y, z). \tag{2}$$

The rules are applied in parallel, and the order of the literals in the body is irrelevant. The transitive closure is computed by iterating the rules over an instance of *Link*, that represents a given graph. For each tuple (α, β) such that $Link(\alpha, \beta)$ holds, the first rule allows to derive $TC(\alpha, \beta)$, and similarly for the second rule. The rules are recursively applied till a least fixpoint is obtained, in this case in a number of steps proportional to the diameter of the graph.

The time complexity can be optimized, by replacing rule (2) by the following rule, which converges in a logarithmic number of steps.

$$TC(x, z) :- TC(x, y); TC(y, z). \tag{3}$$

In this paper, we are interested in networks, where the nodes have initially only the knowledge of their neighbors. The *Link* relation is thus distributed over the network such that each node has only a fragment of it.

The Netlog programs are installed on each node, where they run concurrently. The computation is distributed and the nodes exchange information. The facts deduced from rules can be stored on the node, on which the rules run, or sent to other nodes. The following rules specify *TC* distributively:

$$\updownarrow TC(x, y) :- Link(x, y). \tag{4}$$

$$\updownarrow TC(x, z) :- TC(x, y); TC(y, z). \tag{5}$$

The **affectation operator** in front of rules determines where the results are affected. The effect of "$\downarrow$" is to **store** the results of the rule on the node where it runs; "$\uparrow$", to **push** them to its neighbors; and "$\updownarrow$", to both store and push them.

The previous program computes the transitive closure in a distributed fashion as follows. The results of the rules are both ($\updownarrow$) stored locally and pushed to neighbors. When it converges (after a number of rounds proportional to the diameter in a synchronous system), the transitive closure is distributed over the network, each node deriving in particular the nodes reachable from itself.

Let us consider more closely the semantics of the affectation operators. Assume in the sequel that rule (4) is installed on each node, and let's focus on the recursive rule. In rule (5), it is important that results are both stored and pushed ($\updownarrow$). The following **store rule** would compute paths in the direct neighborhood of each node, without communication.

$$\downarrow TC(x, z) :- TC(x, y); TC(y, z). \tag{6}$$

The next **push rule**, on the contrary, would lead to an infinite loop of communication, with no result stored.

$$\uparrow TC(x, z) :- TC(x, y); TC(y, z). \tag{7}$$

Indeed, $\downarrow$ is the only **"write"** instruction in the language. Facts that are received by a node are only used to trigger rules.

Let us now consider the locations on which the rules run, and where the results are sent. Consider again rule (2). The following rule stores and pushes its result.

$$\updownarrow TC(x, z) :- Link(x, y); TC(y, z). \tag{8}$$

Given that nodes store only their neighbors in *Link*, rule (8) instantiates either x or y by the node Id, say α, on which it is executed. Suppose first that α instantiates x. Then, the node α will store (and push) facts $TC(\alpha, \gamma)$ for any γ, reachable from α. Suppose instead that α instantiates y. Then α stores and pushes facts $TC(\beta, \gamma)$ for β, neighbor of α, and γ reachable from (β through) α.

Such facts although irrelevant for α, can be useful for β, to which they can be sent by the following rule, which **unicast** the facts, using the **destination instruction** "@", on a variable of the head, instead of pushing them to all neighbors.

$$\updownarrow TC(@x, z) :- Link(x, y); TC(y, z). \tag{9}$$

The **destination instruction** can apply to a node *Id*, *all* or *nil*. If the deduced fact contains $@\beta$, $@all$ or $@nil$, then its destination is respectively node β, each node in the network, or each neighbor. If its destination is not a neighbor, we will see in the sequel to which neighbor the message containing the fact is pushed, according to the knowledge the node has of the *Route* relation.

To avoid computing irrelevant facts, it is as well possible to force the computation to take place on the node instantiating x. This is expressed in the following rules, using the **location instruction** "@" in front of a unique variable in the body of the rule, so that this variable is instantiated by the node's Id where the rule runs.

$$\updownarrow TC(x, y) :- Link(@x, y). \tag{10}$$
$$\updownarrow TC(x, z) :- Link(@x, y); TC(y, z). \tag{11}$$

Rules (10) and (11) essentially partition the results of TC on relevant nodes.

Let us now consider routes, which for each destination give the next hop on the path to that destination. The relation $Route(Src, Hop, Dst)$ extends TC, with an attribute for the next hop. Routes are stored in the routing table, *Route*, and are used by the Router for passing messages to their destination. Rules (10) and (11) can be adapted easily to define routes as follows.

$$\updownarrow Route(x, y, y) :- Link(@x, y). \tag{12}$$
$$\updownarrow Route(x, y, z) :- Link(@x, y); Route(y, u, z). \tag{13}$$

As above, assume now that rule (12) is stored on each node. Note that rule (13) is very naive and results in all possible routes. The following rule avoids recomputing routes, when a route is already known.

$$\updownarrow Route(x, y, z) :- Link(@x, y); Route(y, u, z); \neg Route(x, \_, z). \tag{14}$$

It makes use of a **universal literal**, "$\neg Route(x,\_,z)$", which is interpreted by a universal quantification: there is no route from x to destination z, for any value of the next hop. The first route discovered is then stored and pushed.

Netlog also contains standard arithmetic and aggregation functions, as illustrated below. Routes can be compared according to their length for instance. The following program computes such weighted routes.

$$\updownarrow WRoute(x,y,y,1) :- Link(@x,y). \tag{15}$$

$$\updownarrow WRoute(x,y,z,n) :- Link(@x,y); WRoute(y,u,z,n'); \neg WRoute(x,\_,z,\_); n := n'+1. \tag{16}$$

Rule (16) stores the first route discovered and sends it to its neighbors. It uses an **assignment literal** (:=) together with arithmetic operations. Alternatively, the nodes can send the minimal routes, which can be defined using **aggregation** as follows.

$$\downarrow WRoute(x,y,y,1) :- Link(@x,y). \tag{17}$$

$$\updownarrow SLength(x,z,Min(n)) :- WRoute(@x,y,z,n). \tag{18}$$

$$\downarrow WRoute(x,y,z,n) :- Link(@x,y); SLength(y,z,n'); n := n'+1. \tag{19}$$

Rule (18) groups the weighted routes by (Src, Dst), and selects the one with the minimal length. As a side effect, it deletes the facts $SLength(x,z,n')$ with a value $n' > Min(n)$ from the local data store.

We next introduce a construct, the **consumption operator**, !, whose effect is to delete the facts that are used in the body of the rules from the local data store. The effect of the following rule is to delete the oversized *WRoute* facts.

$$\updownarrow SLength(x,z,Min(n)) :- WRoute(@x,y,z,n); !WRoute(@x,y',z,n'); n' > n \tag{20}$$

The consumption operator is the only explicit deletion available in the language, and it applies only to the local store, since the language is local. The above program (rules (17)-(20)) produces the minimal length route between each pair of source and destination. In case of plurality, one route can be chosen non-deterministicaly using the **choice operator**, $\diamond$.

$$\downarrow CRoute(x,\diamond y,z,n) :- SLength(x,z,n); WRoute(@x,y,z,n). \tag{21}$$

Rule (21) groups the routes with minimal length for each pair of source (the node's Id) and destination, and selects one route (next hop) randomly.

Note that the aggregation and the choice operator can be used together in the head of a rule. The following rule chooses a neighbor associated with its degree.

$$Neighbor(@x,\diamond y,\#z) :- Link(x,y); Link(y,z). \tag{22}$$

The variable y is interpreted by a value such that $Link(x,y)$ holds, and the expression $\#z$ is interpreted by the count over all values z such that $Link(y,z)$, for the previously chosen y value.

4 Distributed Fixpoint Semantics

Netlog programs are running on the nodes of the network. They produce facts to store as well as facts to sent to other nodes. Their semantics on one node is defined by fixpoint in a way which is classical for rule-based languages such as Datalog. We extend the fixpoint operators to take all the constructs (arithmetic, aggregation, non-deterministic choice) into account.

We distinguish between two sorts, an uninterpreted sort $(\mathbb{N}, \leq)$, and an arithmetic sort $(\mathbb{R}, \leq, +, \times)$. Assume we are given a set of relations S, called a relational schema, which contains relation *Link*. Given a finite set V of variables, a *valuation* over V is a mapping from V to $\mathbb{N} \cup \mathbb{R}$. Let $Var(r)$ be the set of variables of some rule r over schema S. Let $\mathcal{V}(Var(r))$ be the set of valuations σ over $Var(r)$ which respect the sorts.

Let I be an instance over schema S. The **satisfaction** of the literals in the body of rule r by instance I and valuation σ is defined in a classical way, but for the universal literal, where: $(I, \sigma) \models \neg R(t_1, \ldots, -, \ldots, t_n)$ iff for any constant C, $R(\sigma(t_1), \ldots, C, \ldots, \sigma(t_n)) \notin I$. Assume the body of r, $body_r$, is $L_1, \ldots, L_\ell$. We have $(I, \sigma) \models body_r$ iff $(I, \sigma) \models L_i$, for each $i \in [1, \ell]$.

Now we define the valuation of the head, $head_r$, of rule r. In Netlog, aggregate functions and $\diamond$-operators can only occur in the head of rules. Let $Var^{\cancel{Agg\diamond}}(head_r)$ be the simple variables in the head, which are neither arguments of aggregate functions nor of $\diamond$-operators, and $Var^{\cancel{Agg}}(head_r)$ be the variables in the head which are not arguments of aggregate functions.

Let $\tau \in \mathcal{V}(Var^{\cancel{Agg\diamond}}(head_r))$. We extend τ to $\mathcal{V}(Var(r))$ with respect to interpretation I, as:

$$[\tau]_{I,r} = \{\sigma | \sigma \in \mathcal{V}(Var(r)), \sigma(x) = \tau(x), \text{ for all } x \in dom(\tau), \text{ and } (I, \sigma) \models body_r\}.$$

In the sequel, we assume that $[\tau]_{I,r} \neq \emptyset$. We define $\tau(head_r)$ as follows:

- If $head_r$ contains only simple variables and is of the form $R(x_1, \ldots, x_n)$, $\tau(head_r) = R(\tau(x_1), \ldots, \tau(x_n))$.
- If $head_r$ is of the form $R(x_1, \ldots, x_n, Aggr(y_1), \ldots, Aggr(y_m))$, without $\diamond$-terms, then $\tau(head_r) =$

  $$R(\tau(x_1), \ldots, \tau(x_n), Aggr\{\{\sigma(y_1)|\sigma \in [\tau]_{I,r}\}\}, \ldots, Aggr\{\{\sigma(y_m)|\sigma \in [\tau]_{I,r}\}\}).$$

  where $\{\{\ \}\}$ denotes multi-set and Aggr, an aggregate function on multi-sets.

If the head contains $\diamond$-terms, let $\tau_\diamond \in \mathcal{V}(Var^{\cancel{Agg}}(head_r))$ be a valuation. Similarly, we have $[\tau_\diamond]_{I,r}$ defined as above, and we assume $[\tau_\diamond]_{I,r} \neq \emptyset$.

- If $head_r$ is of the form $R(x_1, \ldots, x_n, Aggr(y_1), \ldots, Aggr(y_m), \diamond(z_1), \ldots, \diamond(z_l))$, with $\diamond$-terms, then $\tau(head_r)$ is an element α of the set:

$$\{R(\tau(x_1), ..., \tau(x_n), Aggr\{\{\sigma'(y_1)|\sigma' \in [\tau_\diamond]_{I,r}\}\}, ..., Aggr\{\{\sigma'(y_m)|\sigma' \in [\tau_\diamond]_{I,r}\}\}, \\ \tau_\diamond(z_1), ..., \tau_\diamond(z_l)) \mid i \in [1, n], \tau_\diamond(x_i) = \tau(x_i)\}.$$

For simplicity, we write: $\tau(head_r) \rightsquigarrow \alpha$, where $\rightsquigarrow$ denotes a non-deterministic mapping.

We can now define the set of positive consequences of a program P over an instance I, $\Delta_P^+(I)$, as well as the set of consumed facts, $\Delta_P^-(I)$. First, the set of the possible derived facts of a program P over an instance I is defined by:

$$Facts_P(I) = \{\tau(head_r) | r \in P, \tau \in \mathcal{V}(Var^{\cancel{Aggr}}(head_r)), [\tau]_{I,r} \neq \emptyset\}.$$

We are interested in subsets of $Facts_P(I)$ which satisfy a functional dependency $x_1, \ldots, x_n \rightarrow z_1, \ldots, z_\ell$, that is those subsets of facts where a single choice was made for all variables of *diamond* operators. Let $\mathcal{P}_P(I)$ be the set of such subsets of $Facts_P(I)$. Then,

$$\Delta_P^+(I) \rightsquigarrow J, \text{ where } J \in \mathcal{P}_P(I);$$

$$\begin{aligned}\Delta_P^-(I) = \{&R(\sigma(t_1), \ldots, \sigma(t_n)) | r \in P, (I, \sigma) \models body_r, !R(t_1, \ldots, t_n) \text{ in } body_r\}\\ \cup\{&R(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_l) | r \in P, head_r =\\ &\quad R(x_1, \ldots, x_n, Aggr(y_1), \ldots, Aggr(y_m), \diamond(z_1), \ldots, \diamond(z_l)),\\ &\quad\quad R(\alpha_1, \ldots, \alpha_n, \beta'_1, \ldots, \beta'_m, \gamma_1, \ldots, \gamma_l) \in \Delta_P^+(I),\\ &\quad\quad\quad R(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_l) \in I\}.\end{aligned}$$

It is not hard to see that $\Delta_P^-(I) \subseteq I$.

We can now introduce the semantics of Netlog programs in a distributed setting. We assume that a program P has been installed on each node of the network. We denote by $P_\downarrow$ the subset of store rules, and $P_\uparrow$ of push rules in P. Note that store-and-push rules belong to both sets.

We monitor the activity, computation and communication, on one node, say α. At each local round, on each node, the program takes as input the local data and the data pushed by other nodes, and produces updated local data, and data to be pushed. The node also forwards messages, that are not used in the local computation. Its interaction with the rest of the network is defined by the *communication function*: $\mathcal{R}^\alpha(\ell)$, which maps ℓ to the set of incoming messages on node α at local round ℓ.

Note that at each local round, the router sorts the incoming messages into two sets $\mathcal{L}^\alpha(\ell)$, of *received facts*, and $\mathcal{F}^\alpha(\ell)$, of *messages to forward* to other nodes depending upon their destination: $\mathcal{L}^\alpha(\ell)$ contains the facts extracted from messages received from other nodes, with destination α, "*all*", or "*nil*". $\mathcal{F}^\alpha(\ell)$ contains the messages received from other nodes, with a destination different from α or destination "*all*", which will be forwarded further to other nodes.

$\mathcal{F}^\alpha(\ell) = \big\{(fact, dest) \big| (fact, dest) \in \mathcal{R}^\alpha(\ell); dest \notin \{\alpha, nil\}.\big\}$;

$\mathcal{L}^\alpha(\ell) = \big\{fact \,\big|\, (fact, dest) \in \mathcal{R}^\alpha(\ell); dest \in \{\alpha, nil, all\}.\big\}$, for $\ell \geq 0$.

The computation relies on two *operators*, associated to program P, (i) for the data to store locally, $\Psi_P^\downarrow$, and (ii) for the data to push to other nodes, $\Psi_P^\uparrow$. They take as input the local instance I, and the received facts L.

– $\Psi_P^\downarrow(I, L) \rightsquigarrow \Delta_{P\downarrow}^+(I \cup L) \cup (I \backslash \Delta_P^-(I \cup L))$ defines the *store operator*, producing facts to store.

– $\Psi_P^{\uparrow}$ defines the *push operator*, producing messages to push:

$$\Psi_P^{\uparrow}(I \cup L) \rightsquigarrow \left\{ (fact, dest) \;\middle|\; \begin{array}{l} fact \in \Delta^{+}_{P^{\uparrow}}(I \cup L);\ \text{and} \\ \text{if } fact \text{ contains an address term } @\beta \text{ or } @all, \\ \text{then resp. } dest = \beta \text{ or } all; \text{otherwise } dest = nil. \end{array} \right\}$$

We use the notation "$\rightsquigarrow$" instead of equality to denote the non-determinism of the result. During one local round, the following computation takes place on each node.

Definition 1. *Given a Netlog program P, an instance I on node α, a set of incoming facts L, a* ***one-round execution*** *of P on α wrt I and L, is given by a sequence $(I_i^\alpha, \mathcal{P}_i^\alpha)_{i \geq 0}$ such that:*

– $I_0^\alpha \rightsquigarrow \Psi_P^{\downarrow}(I, L)$,
– $I_{i+1}^\alpha \rightsquigarrow \Psi_P^{\downarrow}(I_i^\alpha, \emptyset)$, *for* $i \geq 0$;
– $\mathcal{P}_0^\alpha \rightsquigarrow \Psi_P^{\uparrow}(I \cup L)$,
– $\mathcal{P}_{i+1}^\alpha \rightsquigarrow \Psi_P^{\uparrow}(I_i^\alpha) \cup \mathcal{P}_i^\alpha$, *for* $i \geq 0$.

Note that the facts received L are used in the computation, but not stored on the node, while the facts to be sent are accumulated in the $\mathcal{P}_i^\alpha$'s without being used in the computation on α.

The one-round computation of a program on a node consists of any possible one-round execution.

Definition 2. *Given a program P, an instance I on node α, a set of incoming facts L, a* **one-round computation** *of P on α wrt I and L* **terminates** *if all its non-deterministic one-round executions converge to a fixpoint, i.e., every sequence $(I_i^\alpha, \mathcal{P}_i^\alpha)$ has a limit (I^α, P^α) for $i \to \infty$. Such a limit is called a* ***one-round fixpoint of the program*** *P on node α wrt I and L.*

When a local round ℓ starts, the node α has a local instance $\mathcal{I}^\alpha(\ell)$, and has received facts $\mathcal{L}^\alpha(\ell)$, and messages to forward $\mathcal{F}^\alpha(\ell)$. It then starts its computation, and produces a new local instance $\mathcal{I}^\alpha(\ell+1) \rightsquigarrow lim_{i\to\infty} I_i^\alpha$ and a set of messages to push $\mathcal{P}^\alpha(\ell) \rightsquigarrow lim_{i\to\infty}\mathcal{P}_i^\alpha \cup \mathcal{F}^\alpha(\ell)$ if the limits exist.

Let us now consider the communication between nodes. The messages to push are accumulated in $\mathcal{P}^\alpha(\ell)$. Their routes will be computed according to the knowledge node α has of the *Link* and *Route* relations (see the Router description in Section 2).

In the case of synchronous systems without failure, there is an explicit correspondence between the incoming and outgoing sets of messages.

Proposition 1. *For synchronous systems without failure, we have for $l \geq 0$:*
$\mathcal{R}^\alpha(0) = \emptyset$,

$$\mathcal{R}^\alpha(\ell+1) = \left\{ (fact, dest) \;\middle|\; \begin{array}{l} \exists\beta \text{ s.t. } Link(\beta,\alpha) \in I^\beta(\ell); (fact, dest) \in \mathcal{P}^\beta(\ell); \text{and} \\ \text{if } dest \notin \{\alpha, nil, all\}, \text{then } Route(\beta,\alpha,dest) \in I^\beta(\ell) \end{array} \right\}.$$

The proof is straightforward.

In the case of asynchronous systems, the function $\mathcal{R}^\alpha$ depends upon the distributed system, and in general might differ between two executions. The semantics is thus defined up to the system of communication function $\mathcal{R}^\alpha$ for each node α. We next define the termination of programs which relies on the convergence of the sequence of fixpoints.

Definition 3. *Given a program P, running on a network $\mathcal{G}$ with an instance I distributed on each node, and a system of communication function $(\mathcal{R}^\alpha)_{\alpha \in V_\mathcal{G}}$, a* ***computation*** *of P on $\mathcal{G}$ wrt I and the $\mathcal{R}^\alpha$'s* **terminates** *if on each node α, and at each round ℓ all the one-round computations of P converge to a fixpoint, i.e. all sequences $(I_i^\alpha(\ell), \mathcal{P}_i^\alpha(\ell))$ have a limit $(I^\alpha(\ell), \mathcal{P}^\alpha(\ell))$ for $i \to \infty$, and moreover all sequences $(I^\alpha(\ell), \mathcal{P}^\alpha(\ell))$ have a limit $(I^\alpha, \mathcal{P}^\alpha)$ for $\ell \to \infty$. The collection of limits $(I^\alpha)_{\alpha \in V_\mathcal{G}}$ is called a* ***distributed fixpoint of the program*** *P.*

5 Complexity

In this section we investigate the complexity of Netlog programs. Their termination is of course undecidable. Nevertheless, for restricted classes of programs, we can obtain bounds on their complexity. We consider three complexity measures. Two are classical in distributed computing, the distributed time and the message complexity. The last one, the in-node complexity, is generally ignored for distributed systems, but it is interesting in this context since it admits nice bounds as well.

- The *distributed time complexity*, is the maximum number of rounds of any local execution of any node till the termination;
- The *per-node message complexity*, is the maximum number of messages sent by any node till the termination;
- The *per-round in-node computational complexity*, is the time complexity of the in-node computation in one round.

Several factors can cause the non-termination of a program. (i) A program can generate an unbounded number of new values, by using arithmetic functions for instance. Even if the domain in which the program ranges is bounded, (ii) the sequences of instances $I_i^\alpha(\ell)$ can very well not converge at some round ℓ. Or, (iii) the sequences $(I^\alpha(\ell), \mathcal{P}^\alpha(\ell))$ do not have limits.

By controlling these three causes of non-termination, we can obtain well-behaved programs, which admit polynomial complexity bounds. To solve the first problem, range restrictions can be imposed on the variables in the rules to guarantee that they range over some finite set of values. The main problem is to prevent arbitrary recursion over the creation of new values.

A program P is **range-restricted**, if for each input instance, there is a domain of size polynomial in the instance (for a polynomial depending upon P), such that the fixpoint of the program can be computed over this restricted domain, that is with all variables ranging over the restricted domain, while producing the same result. Although undecidable, this property can be enforced by syntactic restrictions.

For lack of space, we only illustrate such restrictions on two examples of Section 3. In rule (16), the fourth attribute of $WRoute$, with variable n, say Lth, is a new-value attribute. The literal $\neg WRoute(x, \_, z, \_)$ in the body of the rule guarantees a functional dependency from (Src, Dst) to Lth, and thus a bound on the value of Lth. In rules (18) and (19), there is a recursion between $SLength$ and $WRoute$. The aggregation function in rule (18) ensures a functional dependency from the first and second attributes of $SLength$ to the third, and a linear bound on these values. It follows that the number of values in $WRoute$ is also linearly bounded.

Let us tackle now the second problem. We say that a program P is inflationary if $I \subseteq \Psi_P^{\downarrow}(I, L)$ for any set of facts L. Programs without consumption nor aggregate function are inflationary. However, this is a very restrictive condition but it can be relaxed, by allowing to replace monotonically, at each iteration of the fixpoint operator, facts with an aggregate attribute (with the aggregated value continuously either increasing or decreasing). Such programs are called **quasi-inflationary**. Rules (18) and (19) define a quasi-inflationary program for instance adding continuously facts in relation $WRoute$ and updating $SLength$ with continuously smaller values.

The third problem can be tackled by **guarded communication**. A push rule is **guarded** if its body can only be instantiated using facts from the local instance, not from the incoming messages. At the syntactic level, this can be enforced easily by forbidding recursion over head relations in $P^{\uparrow}$. Consequently, if L is a set of facts over the head relations of $P^{\uparrow}$, $\Psi_P^{\uparrow}(I \cup L) = \Psi_P^{\uparrow}(I)$. Rule (18) for instance is guarded.

A program is **well-behaved** if it is a range restricted, quasi-inflationary program with guarded communication. We can prove the following result.

Theorem 1. *Well-behaved programs have distributed time complexity, per-node message complexity, and per-round in-node computational complexity polynomial in the size of the input instance.*

Netlog programs can be transformed into equivalent programs which admit more efficient execution. We have considered several aspects of the **optimization** as well as the **adaptive behavior** of programs. First, the implementation of Netlog is based on a **semi-naive evaluation**, which triggers only rules over inputs where one of the relations in the body of the rule has been updated since the previous iteration of the fixpoint. Second, the implemented version of Netlog supports **modules** of rules. Programs are decomposed into distinct modules, which model specific tasks and trigger one another after completing their fixpoints.

6 The Netquest System

The Netquest virtual machine presented in Section 2 has been implemented. It relies on an embedded DBMS, with which the Engine is coupled.

We choose to rely on an embedded DBMS, to simplify the programing of the system, increase its portability, and allow the extension to data intensive applications. This choice is by no means a limitation since an increasing number of small devices have now embedded DBMSs such as smart phones or iMote devices, for which we carried out experiments. The Netlog programs are compiled into SQL queries, which are then loaded on the embedded databases. Our compiler can currently produce queries for either MySQL or SQL Server.

The main component is the Engine which computes the fixpoints I^α and $\mathcal{P}^\alpha$ on each node α. It loads the queries corresponding to a program and runs them against the database, till a fixpoint is reached. Most of the computation is thus performed by the DBMS. The Engine has some additional functionalities, not developed in the present paper, such as timers, necessary for networking protocols. Programs are organized into modules to ease programming. Netquest also uses optimization techniques, such as the triggering of rules by new facts, which avoid unnecessary re-computation, when there are no changes in the input of rules. Netquest also relies on a more complex type system standard for programming languages, and integrates aggregate functions available in SQL.

The router handles the queues of incoming and outgoing messages, and works according to the semantics presented in this paper. This implies to revisit the functionalities of standard routers.

Netquest has been installed and tested over two platforms: the network simulator WSNet [8], as well as a network emulator developed in the project. A large set of protocols from different areas have been programmed in Netlog and tested over these two experimental platforms, while a visualization tool allows to follow the network activity, the communication, as well as the execution of individual rules.

7 Conclusion

Declarative languages for distributed programming are very promising, but they raise technical difficulties. In this paper, we have proposed a new rule-based language, that on one hand is well suited for programming network applications and protocols, but meanwhile admits a well defined semantics, which solves problems raised by previous proposals.

Our objectives are to produce code which is (i) easy to write because it relies on declarative statements; (ii) adaptive, can be compiled into different algorithms depending upon the dynamic context; and (iii) verifiable formally. All these objectives require a formal semantics.

We are currently far though from declarative languages for networking. Indeed, in current proposals, most of the distributed optimization techniques has to be expressed in the rules. We are currently working on automatic translation of rule programs to equivalent programs which are optimized, can adapt to changes in the network, much like query optimization techniques in the context of databases.

We choose to implement Netlog on top of a DBMS, to allow data intensive applications, and increase the portability of the system over heterogeneous devices and networks. Our first experiment on devices are rather conclusive.

Acknowledgments

The authors wish to thank Michel Bauderon, Wenwu QU, Kun SUO, Xin QI, Zhilin WU, and Huimin LIN for fruitful discussions on the language as well as for their strong involvement in the development of the Netquest system.

References

1. Abadi, M., Loo, B.T.: Towards a declarative language and system for secure networking. In: NETB 2007: Proceedings of the 3rd USENIX international workshop on Networking meets databases, pp. 1–6 (2007)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
3. Alonso, G., Kranakis, E., Sawchuk, C., Wattenhofer, R., Widmayer, P.: Probabilistic protocols for node discovery in ad hoc multi-channel broadcast networks. In: Pierre, S., Barbeau, M., Kranakis, E. (eds.) ADHOC-NOW 2003. LNCS, vol. 2865, pp. 104–115. Springer, Heidelberg (2003)
4. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. Wiley Interscience, Hoboken (2004)
5. Bejerano, Y., Breitbart, Y., Garofalakis, M.N., Rastogi, R.: Physical topology discovery for large multi-subnet networks. In: INFOCOM (2003)
6. Bejerano, Y., Breitbart, Y., Orda, A., Rastogi, R., Sprintson, A.: Algorithms for computing qos paths with restoration. IEEE/ACM Trans. Netw. 13(3) (2005)
7. Demers, A.J., Gehrke, J., Rajaraman, R., Trigoni, A., Yao, Y.: The cougar project: a work-in-progress report. SIGMOD Record 32(4), 53–59 (2003)
8. Fournel, N., Fraboulet, A., Chelius, G., Fleury, E., Allard, B., Brevet, O.: Worldsens: from lab to sensor network application development and deployment. In: 6th International Conference on Information Processing in Sensor Networks, IPSN, pp. 551–552 (2007)
9. Jeffery, S.R., Alonso, G., Franklin, M.J., Hong, W., Widom, J.: Declarative support for sensor data cleaning. In: Fishkin, K.P., Schiele, B., Nixon, P., Quigley, A. (eds.) PERVASIVE 2006. LNCS, vol. 3968, pp. 83–100. Springer, Heidelberg (2006)
10. Liu, C., Mao, Y., Oprea, M., Basu, P., Loo, B.T.: A declarative perspective on adaptive manet routing. In: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow, New York, NY, USA, pp. 63–68 (2008)
11. Loo, B.T., Condie, T., Garofalakis, M.N., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking: language, execution and optimization. In: ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA (2006)
12. Loo, B.T., Condie, T., Hellerstein, J.M., Maniatis, P., Roscoe, T., Stoica, I.: Implementing declarative overlays. In: 20th ACM Symposium on Operating Systems Principles, Brighton, UK (2005)

13. Loo, B.T., Hellerstein, J.M., Stoica, I., Ramakrishnan, R.: Declarative routing: extensible routing with declarative queries. In: ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Philadelphia, Pennsylvania, USA (2005)
14. Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: Tinydb: an acquisitional query processing system for sensor networks. ACM Trans. Database Syst. 30(1) (2005)
15. Marron, P.J., Minder, D.: Embedded WiSeNts Research Roadmap. Embedded WiSeNts Consortium (2006)
16. Navarro, J.A., Rybalchenko, A.: Operational semantics for declarative networking. In: Gill, A., Swift, T. (eds.) PADL 2009. LNCS, vol. 5418, pp. 76–90. Springer, Heidelberg (2009)
17. Ramakrishnan, R., Gehrke, J.: Database Management Systems. McGraw-Hill, New York (2003)
18. Ramakrishnan, R., Ullman, J.D.: A survey of deductive database systems. J. Log. Program. 23(2), 125–149 (1995)
19. Srivastava, U., Munagala, K., Widom, J.: Operator placement for in-network stream query processing. In: Twenty-fourth ACM Symposium on Principles of Database Systems, pp. 250–258 (2005)