HAL
open science

Messages with Implicit Destinations as Mobile Agents

Ahmad Ahmad Kassem, Stéphane Grumbach, Stéphane Ubéda

To cite this version:

Ahmad Ahmad Kassem, Stéphane Grumbach, Stéphane Ubéda. Messages with Implicit Destinations as Mobile Agents. AGERE 2012, Oct 2012, Tucson, Arizona, United States. hal-00753177

HAL Id: hal-00753177

https://inria.hal.science/hal-00753177v1

Submitted on 18 Nov 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Messages with Implicit Destinations as Mobile Agents

Ahmad Ahmad-Kassem
Université de Lyon, INRIA
ahmad.ahmad_kassem@inria.fr

Stéphane Grumbach
INRIA
stephane.grumbach@inria.fr

Stéphane Ubéda
INRIA
stephane.ubeda@inria.fr

Abstract

Applications running over decentralized systems, distribute their computation on nodes/agents, which exchange data and services through messages. In many cases, the provenance of the data or service is not relevant, and applications can be optimized by choosing the most efficient solution to obtain them. We introduce a framework which allows messages with intensional destination, which can be seen as restricted mobile agents, specifying the desired service but not the exact node that carries it, leaving to the system the task of evaluating the extensional destination, that is an explicit address for that service. The intensional destinations are defined using queries that are evaluated by other agents while routing. We introduce the Questlog language, which allows to reformulate queries, and express complex strategies to pull distributed data. In addition, intensional addresses offer persistency to dynamic systems with nodes/agents leaving the system. We demonstrate the approach with examples taken from sensor networks, and show some experimental results on the QuestMonitor platform.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—design languages, concurrent, distributed, and parallel languages, very high-level languages

General Terms Design, Languages

Keywords routing by content, intensional destination, declarative networking

1. Introduction

Most of the applications of our everyday life (communication, search, social, etc.), as well as those of our environment (workplace, domotics, transportation, energy, etc.) rely on complex network infrastructure. They require complex distributed algorithms that are difficult to program, require skilled programmers, and offer limited warrantee on their behavior. The dynamics of some networks, with nodes joining or leaving the networks, not to mention the various types of failures increases further the complexity and raises considerable challenges. One of the fundamental barriers to their development is the lack of programming abstraction [28].

Such applications are decentralized and need to adapt dynamically to their environment in a reactive manner. They necessitate a high-level programming paradigm that defines a new level of abstraction and offers features such as interaction, reactivity, autonomy, modularity, and asynchronous communication.

In this paper we propose a framework which allows to program distributed applications in a message-oriented manner, allowing messages as a sort of mobile agents with implicit destinations, that are solved in the network while they are traveling. The destinations of messages are abstracted and defined by queries. The main contribution of the paper is (i) the design of a data centric language, Questlog, which allows to program agents exchanging messages which admit a complex semantics associated to the queries defining their destinations, and (ii) its implementation over a simulation platform. We thus distinguish between intensional destinations, defined by queries, and extensional destinations, defined by node addresses.

The idea of programming messages as active agents has been proposed long ago in [36], where network programs are encapsulated in active messages traveling in the network. It provides a simple way to describe and understand distributed programs. Mobile code such as scripts, applets, and mobile agents is widely used [13, 20]. Our approach though is more restricted. We propose messages that have implicit destinations, that will be solved in the contact of other agents while the message is traveling. Only the query defining the destination is mobile, while the code of the agent that helps solving it, is static.

Recently, the notion of agent-oriented abstractions is proposed in [32], where a new programming paradigm providing a set of abstractions is introduced to simplify the programming of modern applications. However, the mobility of agents as well as the abstraction of the destinations of messages are not supported.

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AGERE! 2012, October 21–22, 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1630-9/12/10. . . $10.00

Messages with implicit destination facilitate the programming of a large class of applications. Publish/subscribe systems constitute a good example of such systems, with publishers who do not have to specify specific receivers, leaving the system matching them. Publish/subscribe systems constitute an appealing paradigm for developing pervasive systems which enable the decoupling of interacting components, separating communication from computation. However, they generally require the use of mediators to match interest with published events. Different forms of publish/-subscribe systems have been proposed.

Topic-based systems [16] rely on the notion of topics, a static scheme with limited expressiveness. *Content-based* systems [16] allow filtering on the content of an event, and only those events that match the filter are delivered to the subscribers. This approach might result in high numbers of topics and potentially redundant events that increase the overhead. *Type-based* systems [16] combine topic-based and content-based system. The idea is to replace the topic classification form by a scheme that filters events according to their type. *Location-based* systems [17] support location-aware communication between participants based on positioning mechanisms, and *context-based* [19] systems capture event context in a modular way.

The major difficulty with publish/subscribe systems rely in the events matching mechanism, the efficient routing of notifications to subscribers, while avoiding useless transmission of notifications that result in an extra level of complexity [29]. Different content-based routing approaches have been proposed to route efficiently notifications by messages based in their content. In [12], a routing scheme is proposed based on a combination of a traditional broadcast protocol and a content-based routing protocol. However, it suffers from a high communication complexity to build spanning trees to send notifications. In [27], authors propose a new method to provide end-to-end reliability based on the publish/subscribe system but with the cost of increased overhead.

To facilitate programming, we propose to use intensional destinations defined by queries, and let active agents in the network solve them on the fly. To do so, active agents have at their disposal programs (local agents) that give a meaning to destinations. More precisely, a *destination* is a pair specified *extensionally*, by the address of the node, and *intensionally*, by a Questlog query.

Declarative query languages have already been used in the context of sensor networks. Several systems such as TinyDB [26] or Cougar [14] 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 [33].

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 [24] and declarative overlays [23]. This approach, known as **declarative networking** is extremely promising. It has been further pursued in [25], where execution techniques for Datalog are proposed. Distributed query languages thus provide new means to express complex network problems such as node discovery [4], route finding, path maintenance with quality of service [9], topology discovery, including physical topology [8], secure networking [1], or adaptive MANET routing [22].

Declarative networking relies on the rule-based languages [5, 6, 31, 35] developed in the field of databases in the 1980's. Questlog follows the trend opened by declarative networking [23, 25]. Declarative languages allow to specify at a high level "what" to do, rather than "how" to do it. They facilitate not only code reuse among systems, but also the extension, and hybridization. It was shown that such languages augmented with communication primitives, allow to express distributed applications and communication protocols with code about two orders of magnitude shorter than imperative programs, and with reasonable execution models. They are more declarative, so facilitate programming, they parallelize well, so facilitate the execution, they manipulate explicitly data structures, so facilitate verification of their properties. Simple Netlog protocols for instance have already been verified [15] using the Coq proof assistant.

Different languages have been proposed such as Overlog [23], NDlog [25], Netlog [18], and Webdamlog [2] for high-level programming abstraction. To our knowledge, however, they all follow the *forward chaining* mechanism. They are very successful in expressing various applications and protocols in proactive mode, but less than in reactive mode.

In contrast to Overlog [23], NDlog [25], Netlog [18], and Webdamlog [2], Questlog has been designed to *pull* data from a network by firing a query. The query is associated with a rule program composed of a set of rules in the form $head$:- $body$ that are evaluated in parallel. The program is installed on the nodes of a network and the evaluation of rules combines *backward* and *forward chaining*. When a node receives a query, it identifies the rules whose head matches the query. If there are such rules, the node applies each of them, that is it generates their body instantiated with the variable substitution imposed by the initial query.

The Questlog language includes complex primitives such as aggregation, non deterministic choice, etc., to facilitate the programming of complex application. Questlog programs are compiled into sets of queries in an SQL dialect, which are loaded on the nodes of the network.

We have developed a system which runs the Questlog programs, and extends the Netquest virtual machine, initially proposed in [18] to evaluate Netlog programs. The new func-

tionalities include (i) a *Questlog Engine* to evaluate queries and programs, and (ii) a *Router* to evaluate intensional destination query that offers resilience of the system under node failure or departure. We demonstrate the approach with examples taken from sensor networks, and show some experimental results on the QuestMonitor [10] platform that allows to interact with a network and visualize the behavior of declarative protocols. The system has been tested on simple networking protocols as well as wireless sensor networks, WSN, applications.

The paper is organized as follows. In the next section, we present motivating examples to explain the use of intensional destinations, and introduce the rules. Questlog, with the language's primitives is presented through examples in Section 3, while its procedural semantics is defined in Section 4. Section 5 is devoted to the implementation on top of the Netquest system, while some experimental results are presented in Section 6.

2. Motivation

We are interested in applications running over networks, with data fragmented over participating nodes, which in general have no knowledge on the location of data. They communicate by exchanging messages with a *payload*, the content of the message, and a *destination*, the final destination. In classical networking approaches, the flow of messages from source nodes to destinations is driven by their addresses (e.g. IP, MAC, etc.) assigned explicitly by the source nodes. In an increasing number of applications however, it is desirable if not necessary to be able to delay the evaluation of the final destination of a message. Examples of such applications include:

- *Distributed hash tables*: A hash function is used to map data items to nodes. Given a value (e.g. Id, address, data, etc.), the hash function produces a key, in general over the domain of identifiers of nodes. The destination can then be for instance the closest node. In Chord [34] or VRR [11] for instance, the nodes are organized in a ring structure, and messages are routed on the ring to increasing or decreasing Ids, till the closest node is reached.
- *Wireless sensor networks*: Such networks consist of large numbers of sensor nodes with limited numbers of sinks, which collect information from sensor nodes. For instance, a sink can collect the positions of nodes which have a temperature greater than some threshold. The sink can thus send messages to subsets of nodes satisfying some property.
- *Publish-subscribe systems*: Users publish services without specifying specific destinations to them, while subscribers express their interest to services, and receive corresponding messages, without knowledge of the publishers. Such systems are handled by appropriate middleware taking care of the messages.
- *Social networks*: Users are organized in network structures with their friends (symmetric links of Facebook) or followers (asymmetric links of Twitter) for instance, with whom they exchange information. Some messages can be addressed to sets of users that are out of the knowledge of users, or difficult to enumerate, such as the friends of their friends. In some social matching networks, it is possible to send notifications of interest to users to be received only by users who have sent in a symmetric manner similar notification of interest to the sender. In this example, the destinations cannot be cleared by the users themselves.

In all these examples, it would make things easier, if it was possible to specify the destination implicitly by a query, defining in an *intensional manner*, the destination of (message) mobile agent, which can be cleared or evaluated while traveling in the network, in an *extensional* manner, as the explicit address of the destination nodes. In Publish-Subscribe systems, this is done by appropriate message oriented middleware. Our objective is to let mobile agent solve intensional destinations.

Let us consider the following more complex example from wireless sensor networks. Consider an application where some sink node monitors the positions of nodes which have, together with their neighbors to avoid individual measurement errors, a temperature higher than some threshold. How to program such queries? How to get neighbors' temperature values dynamically?

We propose a declarative language, *Questlog*, which allows to specify such problems in a rather declarative, data centric manner. For simplicity, we consider a relational model of data, with relations of some fixed schema. *Questlog* is a rule-based language with *rules* of the form:

$$head : -body$$

well-adapted to complex queries as well as to reactive protocols. *Questlog queries* are of a very simple form:

$$?R(x_1, \cdots, x_\ell)$$

where R is a *relation symbol* of arity ℓ, and $x_1, \cdots, x_\ell$ are variables or constants. They are associated to rule programs which define their semantics.

Let us illustrate the language on the previous WSN example. The query can be expressed very simply by a predicate of the form:

$$?WarnPos(v, x, y)$$

where v is a node Id and (x, y) its positions. The meaning of the query is defined by a program (agent), which is used to evaluate it. Let us consider the following program:

$$\uparrow WarnPos(v, x, y) : - \; Pos(v, x, y), \\ Tmp(v, t), t > T. \qquad (1)$$

We assume that each node, say v, stores its location (x, y) as $Pos(v, x, y)$, and its temperature as $Tmp(v, t)$. When the agent on a node, say α, receives a query $?WarnPos(v, x, y)$, it checks if it matches the head of a rule. In this case, it matches Rule (1). Its body, $Pos(v, x, y), Tmp(v, t), t > T$, is instantiated with local data, and the tuples (v, x, y) satisfying the query are produced as answers to the query and sent ($\uparrow$ in front of the rule) to the source of the query.

Let us consider now the more complex example, of nodes v with location (x, y), which have, together with their neighbors, a temperature greater than T. We assume that each node v also stores links to its neighbors, say w, as $Link(v, w)$. The following program defines the new query.

$$\uparrow WarnPos(v, x, y) :- Pos(v, x, y), Tmp(v, t), \\ t > T, \forall w\ Link(v, w), ?HighTmp(@w). \quad (2)$$

$$\uparrow HighTmp(v) :- Tmp(v, t), t > T. \quad (3)$$

The program is interpreted as follows. The query now matches Rule (2). This rule is interpreted as follows. Its body contains facts $Pos(v, x, y)$; $Tmp(v, t)$; as well as an expression: $\forall w\ Link(v, w), ?HighTmp(@w)$. The facts are instantiated locally as above. The new query $?HighTmp(@w)$ is generated for each neighbor w of v (universal quantifier), and sent to each neighbor w (symbol "@" in front of the variable). Suppose that there are nodes β and γ such that $Link(\alpha, \beta)$ and $Link(\alpha, \gamma)$ hold on α. Then α generates two new queries, $?HighTmp(@\beta)$ and $?HighTmp(@\gamma)$, which have to be sent to node β and γ respectively.

Suppose that node β receives the query $?HighTmp(\beta)$. It matches the $head$ of rule (3). This matching leads to $Tmp(\beta, t), t > T$, the body of Rule (3). If the rule is satisfied, then the $head$, $HighTmp(\beta)$, of the rule is generated and sent to α, due to the affectation operator ($\uparrow$), where α is the *origin* of the query. The evaluation of the query $?HighTmp(\gamma)$ is done in a similar fashion on node γ. The results of the initial query $WarnPos(\alpha, x, y)$ will be computed by Rule (2) once *all* the answers to the queries $?HighTmp(@w)$ have been obtained. This is the meaning of the $\forall$ symbol in front of variable w in the body of Rule (2). Then the result is sent to the initial source of the query $?WarnPos(v, x, y)$.

With Questlog, complex applications and protocols can be expressed easily. Consider for instance the following query $?Route(\alpha, d, y, n)$, searching for a next hop y, and a length n, for a route from node α to destination d. The following two rules, Rule (4) and (5), define an *on-demand routing protocol*, which allows to evaluate the initial query $?Route(\alpha, d, y, n)$.

$$\updownarrow Route(x, w, w, 1) :- Link(x, w). \quad (4)$$

$$\updownarrow Route(x, w, z, n+1) :- Link(x, z), \\ ?Route(@z, w, u, n). \quad (5)$$

When node, say α, fires a query $?Route(\alpha, d, y, n)$, the agent on α checks if it matches the head of a rule. The matching results in the body $Link(\alpha, d)$. Two scenarios are then possible. Either, with Rule (4), $Link(\alpha, d)$ holds on node α, and the query can be answered by $Route(\alpha, d, d, 1)$ (d is a neighbor of node α), or Rule (5) generates a body containing a fact $Link(\alpha, z)$ and a new query $?Route(@z, d, u, n)$. Suppose there is a node β such that $Link(\alpha, \beta)$ holds on α. Then Rule (5) generates a new query, $?Route(@\beta, d, u, n)$, which has to be sent to node β.

Suppose now that node β receives the previous query, and that $Link(\beta, d)$ holds on β. The query is evaluated on node β, in a similar fashion. The agent on β can now run Rule (4), and answer the query with $Route(\beta, d, d, 1)$. Two actions are then performed. First, the result is stored in the local store. This is due to the affectation operator ($\downarrow$) in front of Rule (4). Second, the result has to be sent to α, due to the affectation operator ($\uparrow$), where α is the *origin* of the query. When the agent on α receives the answer from the agent on β, it uses again Rule (5), but now in push mode to derive the answer to the query, $Route(\alpha, d, \beta, 2)$, and stores it. As a side effect, intermediate nodes that aggregate answers of subqueries save routes to the destination.

Messages are formed by a payload and a destination. The payload can consist either of data or queries. Similarly, the destination consists of an explicit address, and an implicit address, defined by a query. When the destination of a message is only implicitly known as a query Q, two strategies are possible. Either, Q is included in the destination part of the message, which is then handled only by node satisfying it, or it is included in the payload, and handled by all nodes. We will see in the sequel that it results in different evaluation strategies.

More generally, when the destination as well as the payload are represented by queries, we distinguish in messages between two queries:

- *content-query*: query in the payload,
- *dest-query*: query in the destination.

The *dest-query* might be very simple to solve. Only if a node satisfies the *dest-query*, is it authorized to read and compute the *content-query*. Interestingly, this distinction also allows to optimize the distributed computation of queries.

3. The Questlog language

The language Questlog is used to program the behavior of nodes. 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. This can be done with an agent that communicates periodically with other agents on nodes in its transmission range to update the $Link$ relation upon node joining or leaving. The *Questlog* programs are agents and are installed on each node, where they run con-

currently. The computation is distributed and the nodes exchange information.

Agents interact with each other on the same node. They can query and update the data on the nodes. They interact also with agents on other nodes in the network by producing messages to send on the network. Questlog has been designed to pull data from a network. As it has been shown in Section 2, agents are used in association with a predicate, (e.g. $WarnPos$ in Rule (2) for instance) defining a query, which is solved by running the associated agent.

Before describing the language, let us explain the behavior of queries and agents. The evaluation of the rules combines *backward* and *forward chaining*. Intuitively, when an agent on a node receives a query, it identifies the rules whose head matches the query. If there are such rules, the node applies each of them, that is it generates their body instantiated with the variable substitution imposed by the initial query.

There are two possibilities for the body. The body might be *simple*, with no subquery included, it is then evaluated locally on the node, the answer to the query is deduced by applying the rule in a forward manner, and then sent to the requesting node. If the body is *complex*, with subqueries, then the part of the body without subqueries is evaluated locally. The partial results obtained, lead to partial instantiation of the subqueries, which are then sent to the appropriate nodes. Some bookkeeping is performed to keep track of the initial queries and the corresponding subqueries. When the answers are received, the initial query can be computed, and its answer sent to the requesting node.

As seen in Section 2, the Questlog *queries* are of a very simple form: $?R(x_1, \cdots, x_\ell)$, where R is a relation symbol of arity ℓ, and $x_1, \cdots, x_\ell$ are variables or constants. They are associated to rule *programs* which define their semantics. *Questlog programs* are agents that consist of sets of rules that are executed in parallel.

We introduce Questlog and the primitives of the language through examples. Let us start with routing which is a fundamental functionality for network applications. *On-demand routing* protocols, such as AODV [30], are reactive protocols that flood the network with a route request to find a route from a source to some destination. When the route is found, each node along the route saves locally the next hop to the destination.

We have seen in Section 2, Rules (4) and (5), which express a simple route request. On-demand routing requires some more care though. Indeed, the rules are evaluated in parallel, and the previous two rules could lead at the same time to an answer to the query as well as to useless subqueries propagated to other nodes. For instance, suppose that the $Link$ relation has two facts corresponding to $Link(\alpha, d)$ and $Link(\alpha, \beta)$, where d is the requested destination. Then, Rule (4) leads to a fact $Route(\alpha, d, d, 1)$ as an answer to the query saved locally on α and sent to the source of the query, while Rule (5) leads to a useless subquery $?Route(@\beta, d, u, n)$ sent to neighbor β.

To prevent propagating subqueries when an answer of a query is found locally, we use *negation*. Accordingly, the following routing program, Rule (6) and (7), will be used to evaluate an on-demand routing query. Rule (7) makes use of the literal "$\neg Link(x, w)$" which can be interpreted as follows: there is no link from node x to destination w.

$$\updownarrow Route(x, w, w, 1) : -Link(x, w). \quad (6)$$

$$\updownarrow Route(x, w, z, n+1) : -\neg Link(x, w), Link(x, z), ?Route(@z, w, u, n). \quad (7)$$

When node α fires the query $?Route(\alpha, d, y, n)$, the agent on α checks if it matches the head of a rule. The matching rules, Rule (6) and (7), are loaded, and executed in parallel to evaluate the query. The two rules are instantiated by the instances of the variables in the query. Rule (6) leads to the body $Link(\alpha, d)$. Suppose node d is a neighbor of node α, then Rule (6) is satisfied.

The results of the rules can be either stored locally on the node, or send to other nodes. The arrow in front of the rules specifies it, with $\downarrow$ for local storage, and $\uparrow$ for results sent to the origin of the query, and $\updownarrow$ for both. The deduced answer, $Route(\alpha, d, d, 1)$, is stored in the local data store ($\downarrow$ in front of Rule (6)), and has to be sent ($\uparrow$) to the *origin* of the query. However, Rule (7) generates a body that is not satisfied since the fact $Link(\alpha, d)$ holds on node α.

Intermediate nodes that aggregate answers of subqueries save ($\downarrow$) routes to the destination. It would be interesting to use local knowledge of nodes to reduce the delay and the complexity in both communication and computation. An additional rule is required. The following program with Rules (8), (9), and (10) defines the semantics of the on-demand routing protocol.

$$\uparrow Route(x, w, \diamond y, n) : - Route(x, w, y, n). \quad (8)$$

$$\updownarrow Route(x, w, w, 1) : - Link(x, w), \neg Route(x, w, _, 1). \quad (9)$$

$$\updownarrow Route(x, w, z, n+1) : - \neg Link(x, w), \neg Route(x, w, _, _), Link(x, z), ?Route(@z, w, u, n). \quad (10)$$

Suppose intermediate node γ has a fact, $Route(\gamma, d, \theta, 2)$, saved in the routing table. Rule (8), when receiving the query $?Route(\gamma, d, y, n)$, leads to the body $Route(\gamma, d, y, n)$. The rule is satisfied, then deduced result $Route(\gamma, d, \theta, 2)$ is sent ($\uparrow$) to the source of the query. In case of plurality, one route can be chosen non-deterministicaly using the choice operator, $\diamond$ in front of y. Alternatively, the shortest route can be chosen using aggregation, (e.g. $Route(x, w, y, min(n))$). The evaluation of Rule (9) leads to the body $Link(\gamma, d), \neg Route(\gamma, d, _, 1)$, where underscore means "any value". The fact "$\neg Route(\gamma, d, _, 1)$" is

read as follow: there is no route from γ to d with next hop any value and number of hop is 1. The use of the negation prevents Rules (9) and similarly for Rule (10) to be satisfied when a route is found locally. This concludes of the on-demand routing protocol.

Let us now consider an example of *aggregation query* over sensor networks. Suppose that a tree rooted on a node α has been constructed in the network. Each node, say x, has the relation $Tree(x, y)$ where y is a child of x, and stores a temperature value t in a relation $Tmp(x, t)$. Suppose node α fires the query, $?ResultAvg(\alpha, v)$, asking for the average, v, of the temperature values of deployed sensors in the network. The following program defines its semantics.

$$\downarrow ResultAvg(x, v) :- v := t/n, \quad ?Avg(@x, n, t). \tag{11}$$

$$\uparrow Avg(x, 1, t) :- \neg Tree(x, _), \quad Tmp(x, t). \tag{12}$$

$$\uparrow Avg(x, \Sigma n + 1, \Sigma v + t) :- \forall y\, Tree(x, y), \quad Tmp(x, t), ?Avg(@y, n, v). \tag{13}$$

where $Avg(x, n, t)$ stores the number n of nodes in the tree rooted at x with the sum t of their temperatures. When node α initially fires the query $?ResultAvg(\alpha, v)$, the agent on α checks if it matches the head of a rule. The matching leads by Rule (11) to the body $v := t/n, ?Avg(@\alpha, n, t)$ which gives raise to a new query $?Avg(@\alpha, n, t)$.

The matching of the new query leads either to the body $\neg Tree(x, _), Tmp(x, t)$ of Rule (12) if α is a leaf (i.e. satisfies $\neg Tree(\alpha, _)$), or otherwise to $\forall y\, Tree(\alpha, y)$,$Tmp(\alpha, t)$, $?Avg(@y, n, v)$ by Rule (13). In this later case, a series of queries $?Avg(@y, n, v)$ are generated, which are sent to all the children y of α in the tree. The computation will recursively walk down the tree until reaching the leaf nodes. Suppose nodes γ and λ are two leaf nodes, and node β is their parent. When receiving the query $?Avg(@\gamma, n, v)$ on node γ, Rule (12) is satisfied, and deduced result $Avg(\gamma, 1, t)$ is sent to the source β of the query. Node λ evaluates similarly the query $?Avg(@\lambda, c, v)$.

The results of the query on parent node β will be computed by Rule (13) once *all* the answers to the queries $?Avg(@y, n, v)$ have been obtained, according to the $\forall$ symbol in front of variable y in the body of Rule (13). After the computation, the deduced result $Avg(\beta, \Sigma n + 1, \Sigma v + t)$ is sent ($\uparrow$) to the source of the query. The operator $\sum$ is the function *sum* and it is used to sum the number of children as well as their temperature. Node β increases by 1 the number of nodes, and adds its temperature to the sum of temperatures before sending the result to the source node. Rule (13) will perform a converge-cast of the intermediate results. When agent on node α receives the answer for the query $?Avg(\alpha, n, t)$, by Rule (11), it deduces the average temperature. It uses the assignment literal ":=" together with arithmetic operations (e.g. division "/"). The result is saved locally in the relation $ResultAvg$.

Due to fragile conditions, the measured temperature value of individual sensor nodes might be wrong. To improve the stability of such systems, it is possible to update temperature stored in the Tmp relation on each sensor node with new values such as the average temperature of their neighbors. The query $?Tmp(w, u)$ is fired from some node, say α, with *all* destinations.

$$\downarrow Tmp(x, avg(t)) :- !Tmp(x, t_1), \forall y\, Link(x, y), \quad ?GetNghTmp(@y, t) \tag{14}$$

$$\uparrow GetNghTmp(x, t) :- Tmp(x, t). \tag{15}$$

On each node, say β, the query $?Tmp(\beta, u)$, matches the head of Rule (14) thus leading to the body $!Tmp(\beta, t_1)$, $\forall y\ Link(\beta, y), ?GetNghTmp(@y, u)$. It gives raise to queries of the form $?GetNghTmp(@y, u)$ sent to all neighbors y. Each neighbor upon receiving the query, Rule (15), forwards ($\uparrow$) its own temperature value to the query expeditor β. When all answers (according to $\forall$) are received, Rule (14) continues the evaluation in the push mode, results in the head with a new value t stored ($\downarrow$) on β where t is the average temperature which is defined using aggregation.

The *consumption operator*, !, is used to delete the facts that are used in the body of the rules from local data store. The fact $!Tmp(\beta, t_1)$ is deleted upon evaluating the rule, Rule (14), in the push mode.

In most approaches, all deployed sensor nodes are homogeneous and mono-service, and run one application at a time (e.g. measuring the temperature). It is worth noting that Questlog can express applications and protocols running on heterogeneous devices with mono- or multi-services.

In the next example, we explain the use of destination queries. Assume the sink node sends a message that contains (i) a *content-query* in the payload, and (ii) a *dest-query* in the destination.

In WSN applications, data collection might involve all nodes in a network. However, due to sensor power constraints, it might be preferable [21] that data collection be performed from a subset of nodes only. Assume that the sink node calls sensor nodes that have energy level greater than a threshold as cluster heads, to collect data (e.g. temperature) from their neighbors, aggregate the data, and then send aggregated value with the address of the cluster head to the sink. The sink node sends a message with *content-query* $?Collect(x, s)$ and *dest-query* $?Powerful(x)$ in the network. Suppose that the energy level is saved in the relation $Energy$. The following program defines its semantics:

$$Powerful(x) :- Energy(e), e > n. \tag{16}$$

$$\uparrow Collect(x, avg(s)) :- \forall y\, Link(x, y), \quad ?GetData(@y, s). \tag{17}$$

$$\uparrow GetData(y, h) :- Tmp(x, h). \tag{18}$$

Each sensor node, say ν, upon receiving the message evaluates the *dest-query* $?Powerful(\nu)$ using Rule (16) after matching. If the body $Energy(e), e > n$ is satisfied, then the sensor node ν belongs to the destination, and is now allowed to evaluate the *content-query* $?Collect(\nu, s)$. Otherwise, the message is sent further.

The *content-query* matches the head of Rule (17), which leads to $\forall y\ Link(x, y)$, $?GetData(@y, d)$ that gives raise to queries $?GetData(@y, s)$ sent to all neighbors y. Each neighbor upon receiving the query $?GetData(y, s)$, uses Rule (18) after matching and returns its temperature value to the source ν of the query. When all answers ($\forall$) are received, node ν continue the evaluation of Rule (17) in the push mode, leading to a fact $Collect(\nu, s)$ where s is the average temperature sent ($\uparrow$) to the sink.

4. Procedural Semantics

We make little assumptions on the networks. We consider nodes which communicate by exchanging messages as restricted mobile agents. The communication is asynchronous with no shared memory.

Each node is equipped with an embedded machine (Figure 1) which evaluates the Questlog programs. It is composed of three main components: (i) a *router* to handle the communication with the network; (ii) an *engine* to evaluate the queries; and (iii) a local *data store* to manage the information (Data and Programs) local to the node. The Questlog programs are installed on each node, and used to evaluate Questlog queries fired by the applications or received from other nodes through mobile agents.

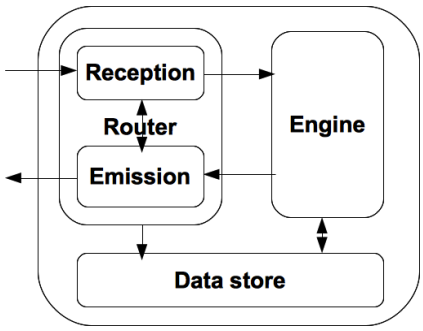

Figure 1. Global architecture of the virtual machine

The evaluation may lead to data (as answers) or subqueries sent to other nodes in the network. Pending queries need to be stored, some bookkeeping is thus performed in the local data store with timeouts. When an answer of a pending query is received, the corresponding query is retrieved and the evaluation is resumed.

4.1 Messages and Routing

We have seen in Section 2 that a message is composed of a payload and a destination. To define precisely the procedural semantics, additional informations in a message are also required. In particular, the source node address, the payload query Id, and the TTL (time-to-live). The TTL is the number of hops that a message is permitted to travel before being discarded by the router. A message has thus the following format:

$$msg = \ < src, qId, payload, dest, ttl >$$

The $payload$ is the content of the message which may contain either a query or data. It has the following format:

$$payload = \ < query \ | \ answer >$$

The $dest$ is the destination of the message. It is composed of both *extensional* and *intensional* destination. The extensional destination is defined by a node address, while the intensional destination is defined by a query. It has the following format:

$$dest = \ < extDest : intDest >$$

The Router is composed of two main modules: (i) Reception module that receives messages from the network, and (ii) Emission module to send messages to other nodes in the network.

When receiving a message, a node first checks the destination. Two cases have to be considered corresponding to extensional and intensional destinations. If the extensional destination is equal to the node address, then the node stores the received message in a local data structure ($BookKeeping$) with a unique Id and a timeout, and transfers the payload to the engine. Otherwise, the node address is not the destination, and the message is transferred to the emission module. For instance, when node β receives the message:

$$msg_1 = \ < \alpha, 4, payload, < \beta : - >, 10 >$$

with address specified extensionally by β which is equal to the node address. Then the message is stored locally ($BookKeeping$), and the $payload$ is transferred to the engine. However, if β receives a message with $dest = < \gamma : - >$, then the message is transferred to the emission module since γ does not match the node address.

Consider now the second case. If the extensional destination is empty, then the router evaluates the intensional destination.

$$msg_2 = \ < \alpha, 4, payload, < - : query >, 10 >$$

The evaluation of the intensional destination $query$ passes through the engine, and the result is a set (σ_{Ans}) of node addresses. When receiving the set of answers, the router checks if the node address is in the set. If so, the router stores locally ($BookKeeping$) the message msg_2, and transfers the payload to the engine to be evaluated. Otherwise, the message is discarded.

At the same time, the initial message msg_2 is transferred to the emission module to be sent to other nodes. It is noteworthy to mention that an alternative strategy could have been used. For instance, instead of transferring the message msg_2 to the emission module, the router could take into consideration the set of answers (e.g. α, β, etc.), encapsulates messages based on msg_2 but with new destinations specified *extensionally* and *intensionally* (e.g. $msg_3.dest =< \alpha : query >, msg_4.dest =< \beta : query >, etc.$), and transfers them to the emission module. The important features of this strategy is: (i) toggling from broadcast mode into unicast mode, and (ii) benefiting from local knowledge of a node. The choice of the strategy can be made by an agent.

The Emission module is used to send messages to other nodes in the network. The router fetches the next hop to the extensional destination from the routing table and sends the message if the next hop is found. Otherwise, the message is discarded. Here again, other strategies can be made and applied as for instance: (i) send the message to neighbors, or (ii) fetch a route to the destination d which in our approach requires to fire the query $?Route(s, d, nh, n)$ while s is the node address, nh is the required next hop, and n is the number of hops, as we have seen in Section 2.

4.2 Computation

A message may contain queries (content-query, dest-query) or data. To evaluate a query, as seen in Figure 2, the mobile agent collaborates with local node agents such as Program agent, Timer agent, Strategy agent, etc. to achieve the task and produce a new mobile agent.

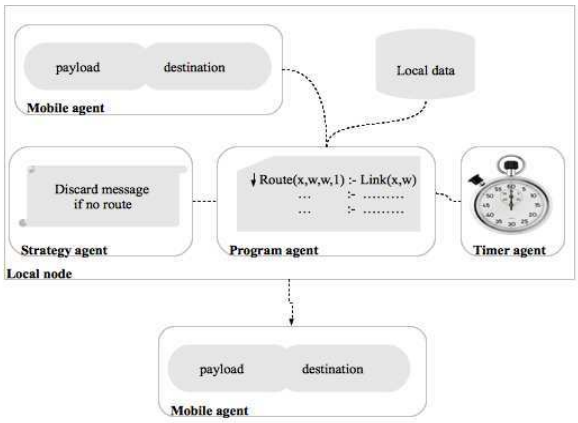

Figure 2. Emission of messages seen as mobile agents

A Timer agent manages program time events and timeout. The timer is defined as a high level specification as follows:

$$Timer(TimerName, Period, Occurrence, ProgName)$$

where the name of the timer, the period to wait before sending an event, the occurrence for repetitive events, and the name of the program are specified.

The engine is in charge of evaluating the received queries and answers. The engine is constructed around two main modules to evaluate them (i) the query module, and (ii) the data module. The query module initiates the evaluation of queries, which may result either in a direct answer to be sent to the query origin, or to subqueries to be sent to other nodes in the network. The data module is used to carry further the computation, and evaluate answers and subsequently pending queries, which may result in an answer saved locally or sent to other nodes.

For each message, its content/dest-query is analyzed and transferred to the corresponding module. When receiving a query, the query module first loads the appropriate rules from the local data store. More precisely, the received query is matched with the $head$ of each rule, and only matching rules are loaded. The rules are then evaluated in parallel. The first step towards their evaluation is the substitution of variables by constants. Rules are instantiated by: (i) potentially the constant values of the received query, and (ii) the local data of the node (where the evaluation is taking place).

Rules can be of two kinds: (i) simple rules, or (ii) complex rules. Simple rules have no subquery in their body, and are evaluated locally on the node. Potentially, local data might satisfy the query, resulting in an answer to be sent to the node source of the query. However, complex rules have subqueries in their body, and their evaluation leads to subqueries propagated to their appropriate destinations.

After the evaluation, two kinds of outputs, either (i) a query, or (ii) an answer can be produced.

- If the result is a query, then the destination to where the query should be sent is extracted from the query. The destination is the instance of the variable prepended by @ in the subquery (e.g. $?Route(@\beta, d, y, n)$). After that, the result is encapsulated in a message which is stored in the local data store ($BookKeeping$), and then transferred to the router. It is worth noting that subquery with destination the source of the initial query can be avoided either by the engine upon evaluation of the initial query (do not send subquery to the source of the initial query) or discarded by the router.
- If the result is an answer, as a fact (e.g. $Route(\beta, d, \gamma, 2)$), then according to the affectation operator of the corresponding rule, the result (i) is stored locally ($\downarrow$), or (ii) sent ($\uparrow$) to the source node, or both stored and sent ($\updownarrow$). The result will be sent in a message, and that requires to collect some information. In particular, the address of the source node of the query is the destination of the message to which the result will be sent. The qId of the message should be the same as the Id of the initial query. The corresponding entry that holds these data is retrieved from the local data store ($BookKeeping$). After that, the message is encapsulated and transferred to the router.

The data module is used to continue the evaluation of pending queries stored locally on a node. When receiving an answer, the data module first loads the appropriate rules from the local data store. More precisely, the engine knows the message qId, communicated by the router, with the payload. The engine matches the received qId with each entry in the $BookKeeping$ data structure, and retrieves the corresponding Questlog rules. Then, the engine evaluates the rules in parallel but now in the push mode. Deduced results are again sent to their appropriate destination exactly as we have seen previously.

5. Implementation

In this section, we present the system which supports the Questlog queries together with their corresponding programs. The network is constituted of nodes that have a unique *identifier*, *Id*, taken from $1, 2, \cdots, n$, where n is the number of nodes. Their communication are based on asynchronous message exchange, and have no shared memory. We make no particular assumption on the nodes/devices which all have the same architecture and the same behavior.

The Questlog programs are transformed into a sort of bytecode that can be smoothly handled. We compile the Questlog programs into an SQL dialect that is executed by the engine.

An SQL query is built for each Questlog operator (query "?", store "↓", push "↑" and deletion "!") in a rule. Consider the following rule witch contains a subquery in the body:

$$\uparrow Route(x, y, z) : - \neg Link(x, y), \\ Link(x, z), ?Route(@z, y, s). \quad (19)$$

The compiler transforms Rule (19) into two SQL queries corresponding to: (i) the results of the operator "?" (body SQL query) to be used in the pull mode for subquery, and (ii) the operator "↑" (head SQL query) to be used in the push mode when receiving an answer in the body of a rule.

The first attribute in the predicate of the head of a rule represents the node address, and it is used as a location specifier. The negation of $Link$ is translated with the SQL subquery into the section $not\ exists$. It is worth noting that the operators "↓" and "!" in a Questlog rule are transformed into an insert and delete SQL query respectively.

Each node is equipped with an embedded machine as we have seen in Section 4. We implemented an extended version of the *Netquest machine* (Figure 3) presented in [18]. Two important functionalities have been introduced (i) a *Router module* to evaluate intensional destinations and to communicate with the network, and (ii) a *Questlog Engine* to execute the Questlog queries and programs.

The Netquest Virtual Machine executes the bytecode, generated by the compiler, and manipulates data and messages. It is working as a daemon in the device, and applications can use it to communicate with other devices on the network. The virtual machine is portable and can be installed in small devices with embedded DMS. A previous implementation was done in iMote sensors [7].

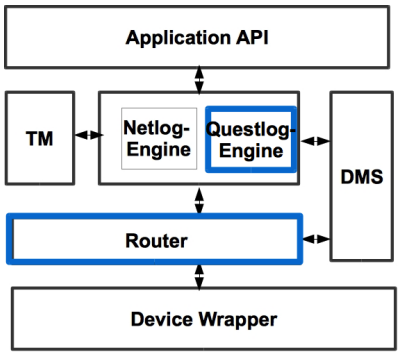

Figure 3. Architecture of the Netquest virtual machine

The Netquest Virtual Machine was initially proposed to evaluate Netlog [18] programs. It is composed of six components; (i) the device wrapper receives and sends data over the network, (ii) the DMS evaluates the bytecode and manipulates data, (ii) the router receives and sends Netquest messages through the device wrapper and chooses the next hop to route a message, (iv) the Netlog engine evalutes Netlog programs by loading the rules and evaluating them through the DMS, (v) the timer manager creates and manipulates timers and manages the time event of the system, and (vi) the application API is in charge of the interaction with local applications.

We next describe the new modules which were added to the Netquest machine, the *Questlog Engine* and the *Router*, together with their functionalities.

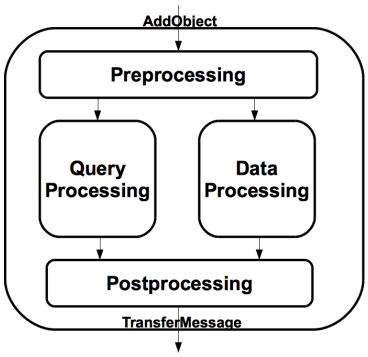

Figure 4. Architecture of the Quesltog engine

The Questlog engine (Figure 4) executes received queries, either from local application or from mobile agents, based on the Questlog programs stored in the local data store. In the proposed model, the message is a mobile agent that may contain a Questlog query. It interacts with local agents that have Questlog programs at their disposition in order to execute the query. They all collaborate to achieve a task. More precisely, the query received by mobile agent is matched with the head of rules of an agent program, that in its turn

may use the timer agent, the routing agent, the neighborhood agent, etc. to finally solve the query.

The engine maintains a data structure, *BookKeeping*, to store queries and answers together with information such as the origin of the query. The Questlog engine is composed of four modules:

1. Preprocessing: This module analyses the incoming messages, in particular the *payload* of the messages. If the content is a query, then the module *query processing* is called to treat the query, otherwise the content is an answer and so the module *data processing* is called.
2. Query processing: This module computes the Questlog queries. For each query, the corresponding rules are retrieved from the local data store. More precisely, a matching operation is performed between the received query and the head of Questlog rules, and then the corresponding SQL queries are retrieved. After that, the SQL queries are executed through the DMS, thus resulting either in an answer for the query, or to the generation of a subquery to be sent to other node. In both cases, the result will be transferred to a *postprocessing* module.
3. Data processing: This module handles data as answers of queries. The corresponding SQL queries are retrieved based on the qId of the received message and the query Id stored in the BookKeeping table. Then, if the corresponding rules contain forall ($\forall$), the SQL queries will not be executed till getting all answers. A local data structure is used to save corresponding received answers. Otherwise, the SQL queries are executed through the DMS and deduced facts are transferred to *postprocessing* module.
4. Postprocessing: This module generates *payloads* in Questlog form by collecting subqueries or facts, fetches their corresponding destinations, encapsulates them in messages, and finally transfers the messages to the router.

The router handles the incoming and outgoing messages through the device wrapper. The specification of the router was described previously in Section 4.

Finally, to facilitate the programming of Questlog programs and to ensure their compilation, we extended the code editor presented in [10] with Questlog syntax coloring and error detection.

6. Simulation Results

The Questlog language is well-adapted to messages with intensional destinations as well as to application queries coming from an API or from external applications running in the network. The queries are on-demand and nodes may enter or leave the network at any time. Our objective here is to monitor the Questlog programs at run time and show their behavior. We thus used a platform that offers these functionalities. The QuestMonitor [10] system is a visualization tool that allows to interact with a network on a 2D graphical interface, and visualize the behavior of declarative protocols. It has three main components:

- The Network Editor: it allows to create groups of nodes, display their status, and install protocols on them;
- The Network Monitor: it allows to visualize different groups of nodes, modify the configuration (e.g. radio range) and interact with the network at run time (e.g. move nodes, delete links, delete nodes);
- The Node Monitor: it exhibits informations about the node selected by the user, allows to monitor the activity, display their data, color nodes and edges, and interact with individual nodes.

We have modified the API of the QuestMonitor system in order to allow a node to send Questlog queries in the network at run time. Figure 5 shows the API where we select a node (e.g. $Node\ 1$) that sends the query, the program to be used (e.g. $OnDemandRouting$), and the appropriate query to be sent in the network (e.g. $?Route(1, 10, y, n)$). Figure 6 shows a small network where node source "1" fires a query $?Route(1, 10, y, n)$ to find a route to the destination "10". The parameters y and n are variables corresponding to the next hop and the number of hops respectively.

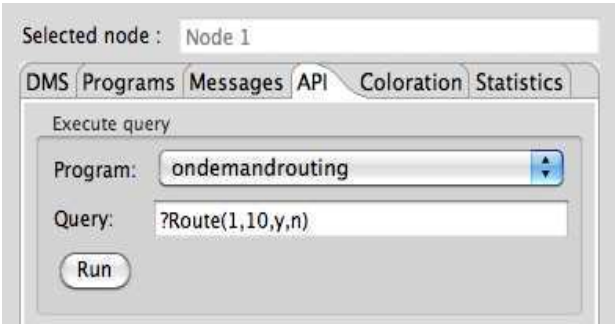

Figure 5. Application programming interface

The Questog programs are installed on each node of the network. Upon running the query $?Route(1, 10, y, n)$ from the API, it is transferred to the engine of node source "1" to be evaluated using, as we have seen in Section 4, Rules (8), (9), and (10). Each node propagates subqueries to neighbors (except neighbor source of the query) if it has no link to the destination.

Figure 6 demonstrates the propagation of queries in messages. The source node sends subqueries to its neighbors (L_1) which in turn repeat the same process (L_i) if no link or route to the destination is found. Intuitively, different routes with different lengths will be received by the source node. The converge-cast of answers by intermediate nodes on the *OnDemandRouting* program follows the same paths of subqueries propagation. Suppose that intermediate nodes have no route to the destination, and that the charge is distributed uniformly over all the nodes in the network, then the first answer received by the source node will be the shortest route.

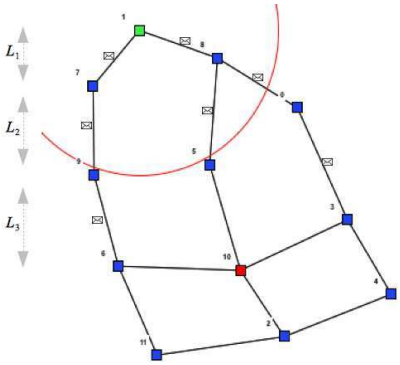

Figure 6. Propagation of queries/answers

In Figure 6 for instance, node 5 is the first node that answers the query.

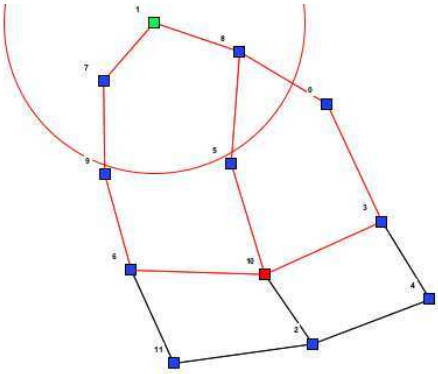

Figure 7. Routes coloration

Intermediate nodes aggregate answers to the source of the query. When receiving the answers, the source node 1 stores their discovered routes in the routing table as seen in Figure 8. Each time a route is built, it will be colored using the coloration feature of QuestMonitor, Figure 7. That allows us to visualize the behavior of declarative network protocols upon link or node failure or departure through direct interaction with the network. In addition, the tab "Statistics" in Figure 8 calculates the number and the kind of queries executed on a node, and results on an average bound of complexity in communication and computation.

7. Conclusion

We have developed a setting which offers messages with implicit destinations, which can be seen as mobile agents, with limited mobile code. They are solved when meeting local agents which have the corresponding code and data to find the best available destination. They ease programming complex applications where the network is used as an active middleware. We proposed a data-centric language, Questlog, that allows to handle intensional destinations as queries and

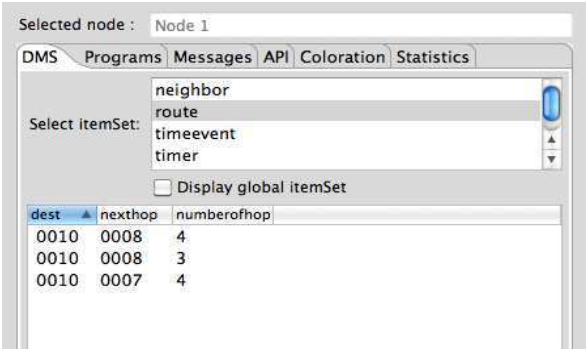

Figure 8. Visualization of ItemSet route

program complex strategies to evaluate them. We have illustrated the language over classical networking protocols, such as routing, and are currently developing sensor network applications as well as social network functionalities including communication, matching, games, etc. The operational semantics of Questlog has been implemented over the Netquest system, and we ran simple examples over the QuestMonitor platform, whose API has been extended to support interactive queries, and to visualize the execution of programs.

We are currently experimenting with the different programming strategies offered by intensional destinations, as well as studying the resulting overhead. These strategies allow to balance the request between the payload and the destination queries, leading to different evaluation schemes. In particular the use of intensional destination can offer persistence to data sent to nodes which have disappeared, and can be rerouted by reevaluating the intensional destination. We have demonstrated such techniques in another context in [3]. Social networks offer challenging reachability problems that we plan to address using this framework in the near future.

Acknowledgments

The authors would like to thank Eric Bellemon for his contribution to the first implementation prototype of the system [10], as well as Christophe Bobineau, Christine Collet, and Fuda Ma, for fruitful discussions. This work has been supported by the Agence Nationale de la Recherche, under grant ANR-09-BLAN-0131-01.

References

[1] M. Abadi and B. T. Loo. Towards a declarative language and system for secure networking. In *Proc. NETB'07*, pages 1–6. USENIX Association, 2007.

[2] S. Abiteboul, M. Bienvenu, A. Galland, and E. Antoine. A rule-based language for web data management. In *PODS*, pages 293–304, 2011.

[3] A. Ahmad-Kassem, E. Bellemon, and S. Grumbach. Seamless distribution of data centric applications through declarative overlays. *BDA'11*, October 2011.

[4] G. Alonso, E. Kranakis, C. Sawchuk, R. Wattenhofer, and P. Widmayer. Probabilistic protocols for node discovery in ad hoc multi-channel broadcast networks. In *Proc. ADHOC-NOW'03*, 2003.

[5] F. Bancilhon. Naive evaluation of recursively defined relations. In *On knowledge base management systems: integrating artificial intelligence and database technologies*, 1986.

[6] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, New York, NY, USA, 1986. ACM. ISBN 0-89791-179-2.

[7] M. Bauderon, S. Grumbach, D. Gu, X. Qi, W. Qu, K. Suo, and Y. Zhang. Programming imote networks made easy. In *The Fourth International Conference on Sensor Technologies and Applications*, pages 539–544. IEEE Computer Society, 2010.

[8] Y. Bejerano, Y. Breitbart, M. N. Garofalakis, and R. Rastogi. Physical topology discovery for large multi-subnet networks. In *Proc. INFOCOM'03*, 2003.

[9] Y. Bejerano, Y. Breitbart, A. Orda, R. Rastogi, and A. Sprintson. Algorithms for computing qos paths with restoration. *IEEE/ACM Trans. Netw.*, 13(3), 2005.

[10] E. Bellemon, V. Dubosclard, S. Grumbach, and K. Suo. Questmonitor: A visualization platform for declarative network protocols. In *MSV 2011: The 8th International Conference on Modeling, Simulation and Visualization Methods, Las Vegas, USA*, 2011.

[11] M. Caesar, M. Castro, E. B. Nightingale, G. O'Shea, and A. Rowstron. Virtual ring routing: network routing inspired by dhts. *SIGCOMM Comput. Commun. Rev.*, 36:351–362, 2006. ISSN 0146-4833.

[12] A. Carzaniga, M. J. Rutherford, and A. L. Wolf. A routing scheme for content-based networking. In *INFOCOM*, 2004.

[13] B. Chen, H. H. Cheng, and J. Palen. Mobile-c: a mobile agent platform for mobile c-c++ agents. *Softw. Pract. Exper.*, 36(15):1711–1733, Dec. 2006. ISSN 0038-0644. doi: 10.1002/spe.v36:15. URL `http://dx.doi.org/10.1002/spe.v36:15`.

[14] A. J. Demers, J. Gehrke, R. Rajaraman, A. Trigoni, and Y. Yao. The cougar project: a work-in-progress report. *SIGMOD Record*, 32(4):53–59, 2003.

[15] Y. Deng, S. Grumbach, and J.-F. Monin. A framework for verifying data-centric protocols. In *FORTE 2011: The 31th IFIP International Conference on FORmal TEchniques for Networked and Distributed Systems*, Reykjavik, Iceland, 2011.

[16] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35 (2):114–131, 2003.

[17] P. T. Eugster, B. Garbinato, and A. Holzer. Location-based publish/subscribe. In *NCA*, pages 279–282, 2005.

[18] S. Grumbach and F. Wang. Netlog, a rule-based language for distributed programming. In *PADL'10, Twelfth International Symposium on Practical Aspects of Declarative Languages, Madrid, Spain*, 2010.

[19] A. Holzer, L. Ziarek, K. R. Jayaram, and P. Eugster. Putting events in context: aspects for event-based distributed programming. In *AOSD*, pages 241–252, 2011.

[20] D. Kotz, R. S. Gray, and D. Rus. Mobile agents: Future directions for mobile agent research. *IEEE Distributed Systems Online*, 3(8), 2002.

[21] L. Kulik, E. Tanin, and M. Umer. Efficient data collection and selective queries in sensor networks. In *GSN*, pages 25–44, 2006.

[22] C. Liu, Y. Mao, M. Oprea, P. Basu, and B. T. Loo. A declarative perspective on adaptive manet routing. In *Proc. PRESTO '08*, pages 63–68. ACM, 2008.

[23] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proc. SOSP'05*, 2005.

[24] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *Proc. ACM SIGCOMM '05*, 2005.

[25] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *Proc. ACM SIGMOD'06*, 2006.

[26] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30, 2005.

[27] A. Malekpour, A. Carzaniga, F. Pedone, and G. T. Carughi. End-to-end reliability for best-effort content-based publish/-subscribe networks. In *DEBS*, pages 207–218, 2011.

[28] P. J. Marron and D. Minder. *Embedded WiSeNts Research Roadmap*. Embedded WiSeNts Consortium, 2006.

[29] J. L. Martins and S. Duarte. Routing algorithms for content-based publish/subscribe systems. *IEEE Communications Surveys and Tutorials*, 01 2010.

[30] C. E. Perkins. Ad-hoc on-demand distance vector routing. In *In Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, 1999.

[31] R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23:125–149, 1993.

[32] A. Ricci and A. Santi. Designing a general-purpose programming language based on agent-oriented abstractions: the simpal project. In *SPLASH Workshops*, pages 159–170, 2011.

[33] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *Proc. POCS'05*, pages 250–258, 2005.

[34] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31, 2001.

[35] L. Vieille. Recursive axioms in deductive databases: The query/subquery approach. In *Expert Database Conf.*, pages 253–267, 1986.

[36] D. W. Wall. Messages as active agents. In *POPL*, pages 34–39, 1982.