



Programming networks with intensional destinations

Ahmad Ahmad Kassem

► To cite this version:

Ahmad Ahmad Kassem. Programming networks with intensional destinations. Computation and Language [cs.CL]. INSA de Lyon, 2013. English. NNT : 2013ISAL0113 . tel-01127033

HAL Id: tel-01127033

<https://theses.hal.science/tel-01127033>

Submitted on 6 Mar 2015

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.

THÈSE

Programming Networks
with Intensional Destinations

Présentée devant

L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE LYON

Pour l'obtention du

GRADE DE DOCTEUR

École doctorale : Informatique et Mathématiques de Lyon

Par

Ahmad AHMAD KASSEM

Soutenue le 04 novembre 2013

Devant la commission d'examen

Jury

Béatrice FINANCE	MCF, HDR, Université de Versailles St Quentin	Rapporteur
Michel BAUDERON	Professeur des universités, Université Bordeaux 1	Rapporteur
Christine COLLET	Professeur des universités, Grenoble INP	Présidente
Letizia TANCA	Professeur des universités, Ecole Polytechnique de Milan	Examinateur
Hassan AIT-KACI	Professeur des universités, Université Claude Bernard Lyon 1	Examinateur
Stéphane GRUMBACH	Directeur de recherche, INRIA - INSA de Lyon	Co-Directeur
Stéphane UBÉDA	Professeur des universités, INRIA - INSA de Lyon	Co-Directeur

Les travaux présentés dans ce mémoire ont été réalisés au laboratoire CITI - INSA Lyon et à temps partiel au laboratoire Franco-Chinois LIAMA en Chine, et financés par le projet UBIQUEST ANR-09-BLAN-0131-01

*A mes parents,
A mes frères et soeurs,
A toute ma famille et tous mes amis*

Acknowledgments

This thesis was realized in the Centre of Innovation in Telecommunications and Integration of service (CITI) of INSA Lyon, and partial time in the Sino-French laboratory for Computer Science, Automation and Applied Mathematics (LIAMA) of INRIA and CASIA (Chinese Academy of Sciences, Institute of Automation) at China. This thesis is supported and financed by an ANR project, UBIQUEST ANR-09-BLAN-0131-01, which involves the three research laboratories CITI, LIG of Grenoble, and LIAMA.

First and foremost, I would like to extend my sincere thanks to my advisors Dr. Stéphane GRUMBACH and Prof. Stéphane UBEDA for their support, patience and trust. During all these years, they have consistently helped me to improve my weak aspects in research. I wholeheartedly appreciate their time which they have invested that made this thesis work a success.

I would like to thank the jury members, Prof. Christine COLLET, Dr. Béatrice FINANCE, Prof. Letizia TANCA, Prof. Hassan AIT-KACI, and Prof. Michel BAUDERON for making the effort to read and review this thesis, for taking a long trip to attend the defense, and for giving me useful feedback. I am particularly grateful for the valuable and constructive suggestions given by Dr. Béatrice FINANCE during the correction of this research work.

I take this opportunity to thank my dear friends and colleagues in CITI, LIAMA and LIG laboratories for good, friendly, and professional, atmosphere all these years. I specially would like to thank IT and administrative staff at CITI and LIAMA for their instant support whenever I asked for one.

Finally, I would like to extend my special gratitude to my parents, family, and friends in France and back home in Lebanon. Their support has always helped me to bounce back whenever I felt low.

Abstract

Distributed programming is a challenging task. It has tremendously gained importance with the wide development of networks, which support an exponentially increasing number of applications. Distributed systems provide functionalities that are ensured by nodes which form a network and exchange data and services possibly through messages. The provenance of the service is often not relevant, while its reliability is essential. Our aim is to provide a new communication model which allows to specify intensionally *what* service is needed as opposed to *which* nodes provide it. The intensional specification of exchanges offers a potential to facilitate distributed programming, to provide persistence of data in messages and resilience of systems, that constitute the topic of this thesis. We propose a framework that supports messages with intensional destinations, which are evaluated only on the fly while the messages are traveling. We introduce a rule-based language, Questlog, to handle the intensional destinations. In contrast to existing network rule-based languages, which like Datalog follow the push mode, Questlog allows to express complex strategies to recursively retrieve distributed data in pull mode. The language runs over a virtual machine which relies on a DBMS. We demonstrate the approach with examples taken from two domains: (i) data-centric architectures, where a class of restricted client-server applications are seamlessly distributed over peer-to-peer systems based on a DHT, and (ii) wireless sensor networks, where a virtual clustering protocol is proposed to aggregate data, in which cluster heads are elected using intensional destinations. Our simulations on the QuestMonitor platform demonstrate that this approach offers simplicity and modularity to protocols, as well as an increased reliability.

Keywords: Programming abstraction; Declarative languages; Distributed systems

Résumé

La programmation distribuée est une tâche difficile. Elle a énormément gagné en importance avec le développement des réseaux qui supportent un nombre croissant exponentiellement d'applications. Les systèmes distribués fournissent des fonctionnalités assurées par les nœuds qui forment un réseau et échangent des données et services, éventuellement par le biais de messages. La provenance du service n'est souvent pas pertinente, alors que sa fiabilité est essentielle. Notre objectif est de fournir un nouveau modèle de communication qui permet de spécifier intentionnellement lequel service est demandé, et non les nœuds qui le fournissent. Cette spécification intentionnelle des échanges offre un potentiel pour faciliter la programmation distribuée, garantir la persistance des données dans les messages et la résilience des systèmes, qui constituent le sujet de cette thèse. Nous proposons donc un cadre qui supporte des messages avec destinations intentionnelles, qui sont évaluées uniquement à la volée au fur et à mesure du déplacement des messages. Nous introduisons un langage, Questlog, qui gère les destinations intentionnelles. Contrairement aux langages à base de règles existants pour les réseaux, comme Datalog, qui suivent le mode "*push*", Questlog permet d'exprimer des stratégies complexes afin de récupérer de manière récursive des données distribuées en mode "*pull*". Le langage fonctionne sur une machine virtuelle qui s'appuie sur un SGBD. Nous démontrons l'approche avec des exemples pris dans deux domaines: (i) les architectures orientées données, où une classe restreinte d'applications client-serveur sont distribuées de manière transparente sur les systèmes pair-à-pair basés sur une DHT, (ii) les réseaux de capteurs sans fil, où un protocole de groupement des nœuds en clusters virtuels est proposé pour agréger les données. Dans ce protocole, les chefs des clusters sont élus à l'aide des destinations intentionnelles. Nos simulations sur la plate-forme QuestMonitor montrent que cette approche offre une simplicité, une modularité aux protocoles, ainsi qu'une fiabilité accrue.

Mots clés: Abstraction de programmation; Langages déclaratifs; Systèmes distribués

Contents

1	Introduction	2
2	State of the Art	8
	Introduction	9
2.1	Declarative Programming	9
2.1.1	SQL-like Query Languages	10
2.1.2	Rule-based Languages	11
2.2	Distributed Algorithms	19
2.2.1	Unstructured Peer-to-Peer Systems	20
2.2.2	Structured Peer-to-Peer Systems	22
2.3	Routing Methodologies	24
2.3.1	Address-based Routing	24
2.3.2	Content-based Routing	25
	Conclusion	27
3	Extensional and Intensional Destinations	28
	Introduction	29
3.1	Message Model	30
3.2	Destination Execution Priority Order	31
3.2.1	Extensional destination higher priority order	31
3.2.2	Intensional destination higher priority order	32
3.3	Intensional Destination Strategies	34
3.3.1	Message decision before processing	34
3.3.2	Message decision after processing	34
3.4	Intensional Destination Specification	35
3.4.1	Intensional destination as SQL query	35
3.4.2	Intensional destination as Questlog query	36
	Conclusion	37
4	Seamless Distribution of Client/Server Applications	38
	Introduction	39
4.1	Client/Server Application	41
4.1.1	Considered Applications	41
4.1.2	Restrictions on Applications	42
4.1.3	Online Multi-player Game Application Example	42
4.2	Distribution Model	45
4.2.1	Distributed Hash Tables	45
4.2.2	Data Distribution	46
4.2.3	Query Distribution	48
4.3	The Netlog language for distributed protocols	49

Contents

4.4	Data centric overlays	52
4.4.1	Distributed lookup	52
4.4.2	Data replication	57
4.4.3	Routing	60
4.5	A Distributed Server for a multiplayer game	62
	Conclusion	64
5	The Questlog Language	65
	Introduction	66
5.1	The Language Questlog	68
5.1.1	The syntax	68
5.1.2	Examples of programs	70
5.2	Procedural Semantics	74
5.2.1	Messages and routing	75
5.2.2	Computation	76
5.2.3	Program execution	78
5.3	Questlog Grammar	80
	Conclusion	83
6	Processing Questlog Programs	84
	Introduction	85
6.1	Data Structures	86
6.1.1	Program structure	86
6.1.2	Predefined data structures for programs	87
6.1.3	Predefined data structures for networks	89
6.1.4	Predefined data structures for system	90
6.2	Questlog Compiler	92
6.3	System Architecture	99
6.3.1	Router	100
6.3.2	Questlog Engine	102
6.3.3	Application Programming Interface and Code Editor	105
	Conclusion	107
7	Protocols with Intensional Destinations	108
	Introduction	109
7.1	Motivation	110
7.2	Data Collection using Questlog	114
7.2.1	Sensor Data Collection	114
7.2.2	One-hop Data Aggregation	114
7.3	Cluster-based Data Aggregation	115
7.3.1	Dynamic Intensional Clustering	116
7.3.2	Aggregated Data Transfer	121
7.4	Experiments over QuestMonitor	124
7.4.1	Load Balancing	124
7.4.2	Dynamic Adaptation	126
7.4.3	Characteristics of Clusters	127
7.4.4	Particular Case	127
7.5	Discussion	128

Contents

Conclusion	129
8 Ubiquest/Netquest Systems and Experiments	130
Introduction	131
8.1 Ubiquest System	131
8.1.1 Data Structures and Languages	132
8.1.2 Ubiquest API	134
8.1.3 Ubiquest Engines	135
8.1.4 Local BMS	136
8.2 Experimentation and Validation	137
8.2.1 Ubiquest Simulation Platform	137
8.2.2 The Results	145
Conclusion	149
9 Conclusion	150
Bibliography	154

List of Figures

3.1	Message model	31
3.2	Virtual ring on physical network topology	33
3.3	Cluster-based wireless sensor network	33
4.1	Data replication techniques	46
4.2	A Chord ring	53
4.3	A virtual ring	57
4.4	Ring before the departure of node 7	60
4.5	Ring after the departure of node 7	60
4.6	Network before moving node 9	62
4.7	Network after moving node 9	62
4.8	Node 7 joins the game	63
5.1	The node architecture	75
5.2	Propagation of subqueries and converge-cast of intermediate answers	79
6.1	Questog program template	86
6.2	Compiler architecture	93
6.3	Netquest virtual machine architecture	99
6.4	Message format	100
6.5	The Questlog Engine	102
6.6	Overview of the QuestMonitor user interface	105
6.7	Application programming interface	106
6.8	Code editor	106
6.9	Compilation results	107
7.1	A topology example	118
7.2	Clustering of trees where roots (cluster heads) are shown as white nodes and branches are shown as arrows	118
7.3	A sensor node architecture	119
7.4	Dynamic cluster-based data aggregation	121
7.5	A simplified schematic for on-demand data transfer	122
7.6	Dynamic adaptation of cluster heads upon topology changes	125
7.7	Dynamic clustering adaptation	126
7.8	A particular topology	128
8.1	An Ubiquet system	132
8.2	Ubiquet virtual machine	133
8.3	Ubiquet Message structure	134
8.4	A query plan	136
8.5	Ubiquet node components	137

List of Figures

8.6	Ubiquest simulation platform user interface	138
8.7	Create group of nodes	138
8.8	Display nodes	139
8.9	Install rule-based program	139
8.10	Network graphical window	139
8.11	Logs window	140
8.12	DMS tab	140
8.13	Programs tab	141
8.14	API tab	141
8.15	DLAQL tab	141
8.16	Case Base tab	142
8.17	Coloring a network path	142
8.18	Coloration tab	143
8.19	Statistics tab	143
8.20	Messages tab	144
8.21	Rule programs code editor interface	144
8.22	A network with DSDV	145
8.23	A network with Tree protocol	147
8.24	A network with Mobile Client at position p_1	148
8.25	A network with Mobile Client at position p_2	148
8.26	Propagation of queries	149
8.27	Visualization of ItemSet route	149
8.28	Routes coloration	149

List of Tables

2.1	Schemas of the <i>transitive closure</i> program	13
2.2	Schemas of the <i>OverLog ping-pong</i> program	14
2.3	Schemas of the <i>NDlog shortest-path</i> program	15
2.4	Schemas of the <i>Netlog routing</i> program	16
2.5	Schemas of the <i>Webdamlog greetings</i> program	17
2.6	Summary of languages characteristics (primitives, properties and system)	18
3.1	Data structure for intensional destination	35
4.1	Main schemas of the <i>Chord</i> protocol	53
4.2	Schemas of the <i>ring</i> protocol	56
4.3	Schemas of the <i>Chord extension</i> for data replication	58
4.4	Schemas of the data replication protocol	59
4.5	Global view of data before the departure of node 7	60
4.6	Global view of data after the departure of node 7	60
4.7	Schemas of the <i>DSDV-like</i> routing protocol	61
4.8	Monitoring route to destination 1 before moving node 9	62
4.9	Monitoring route to destination 1 after moving node 9	62
5.1	Schemas of the <i>on-demand routing</i> protocol	71
5.2	Schemas of the <i>aggregation query</i> program	72
5.3	Schemas of the <i>temperature update</i> program	73
5.4	EBNF main notations	81
5.5	Equivalent notations in Questlog syntax and grammar	81
6.1	Program	87
6.2	Metadata	87
6.3	Timer	88
6.4	Questlog pull rules	88
6.5	Questlog push rules	88
6.6	Questlog variable mapping	89
6.7	Neighborhood table	89
6.8	Routing table	90
6.9	Query store	90
6.10	Reception bookKeeping	91
6.11	Payload bookKeeping	91
6.12	Transmission bookKeeping	91
6.13	Intensional destination bookKeeping	92
6.14	Control for all answers (ControlAnswersForAll)	92
6.15	Questlog to SQL data type conversion	95

List of Tables

7.1	Schemas of the <i>sensor data collection</i> program	114
7.2	Schemas of the <i>one-hop data aggregation</i> program	115
7.3	Neighborhood table	117
7.4	Parameters of simulation over QuestMonitor	127
7.5	Characteristics of clusters over QuestMonitor	127
8.1	Schemas of the <i>DSDV-like</i> routing protocol	146
8.2	Schemas of the <i>Mobile Client</i> protocol	147

Introduction

Distributed programming is undergoing a revolution. It has tremendously gained in importance in recent years. Distributed programming though is a challenging task. One of the fundamental barriers to the development of distributed algorithms is the lack of programming abstraction [109]. The main objective of this thesis is to facilitate distributed programming and to provide algorithms more likely to be correct, verifiable, extensible, with a high level of resilience and persistence. We propose a framework that allows messages with intensional destinations. These destinations are specified by some selection criteria which might be simple, executed locally, or more complex, executed globally. The selection criteria can be expressed by either an imperative language, or a declarative language that provides a higher level of abstraction. We propose such a declarative language which allows to express complex strategies to pull distributed data. This framework is our best-effort towards simplifying the expression of distributed algorithms, with messages treated while routed, and providing persistence to data traveling as well as resilience to distributed systems.

Distributed systems are more and more present nowadays. This is due to the technological advances, starting around the mid-1980s, in the domain of electronics and communication. On one hand, the development of microprocessors enabled an explosive growth of high power, low cost computing. The rise of multi-core microprocessors which, in an attempt to preserve Moore's law, palliate the bound on clock speed, by augmenting the number of cores on chips and parallelizing the computation. Microprocessors are omnipresent nowadays in different entities such as cars, sensors, personal digital assistants (PDAs), mobile phones, wearable computers, airplanes, pacemakers, etc.

On the other hand, wired and wireless communication technologies have led to a major revolution in our societies. On one side, the exponential growth of the Internet has dramatically changed the way we communicate and has enabled the creation of new social structures in the form of virtual communities. With its official protocol standards [140], the Internet has had a tremendous impact on culture and commerce, including the rise of instant communication by email, instant messaging, phone calls (Voice over Internet Protocol), video calls, and the World Wide Web [159] with its discussion forums, blogs, social networking, and online shopping sites. On the other side, wireless technologies have evolved beyond recognition. They use special kinds of devices (e.g. transmitters, receivers, etc.) that utilize electromagnetic waves (radio waves, microwaves, etc.) which carry signals over the entire communication path to transfer information. They are widely used nowadays in different fields such as healthcare, mobile phones, transportation, global positioning system, education, etc.

These developments allow sets of independent entities [151] to be connected with wired and/or wireless technologies, as a network, to communicate, access and share resources such as printers, data, files, Web pages, etc., as well as to do computation. As a result, the wide progress of networks increases tremendously the number of (distributed) applications in many domains (e.g. search, social, communication, workplace, domotics, transportation, energy, healthcare, etc.).

Features of distributed systems

The increasing number of distributed systems and applications has led to growing demands for fundamental properties such as openness, scalability, reliability, persistence, resilience, security, heterogeneity, concurrency, transparency, failure handling, and quality of service. Distributed systems need to simplify for users (and applications) the way to access remote resources and to share them in a controlled and efficient way. Sharing resources is done in a concurrent and cooperative way, and often has to be transparent to the users. As connectivity and sharing increase, security is becoming more and more important to protect information and resources. In addition differences between the various entities and the ways in which they communicate must be hidden from the users. The same holds for the internal organization of the distributed system [151] which should be relatively easy to expand. Users and applications need to interact with a distributed system in a consistent and uniform way, regardless of where and when interaction takes place. A distributed system should be continuously available, although perhaps some parts may be temporarily out of order. Users and applications should not notice that parts are being replaced or fixed, or that new parts are added to serve more users or applications.

Designing and building a distributed system that supports all these features is extremely difficult or even impossible. The features vary on the requirements of the intended system with their associated applications.

Difficulties and challenges of distributed programming

Distributed systems and applications encompass many of the most significant technological developments of recent years (e.g. environmental management, healthcare, creative industries and entertainment, transport and logistics, the information society). They require complex distributed algorithms that are difficult to program [98, 93], necessitate skilled programmers, and offer limited warrantee on their behavior. A programmer might understand what individual entities do [113] and how they react, but it is difficult to understand the behavior of entities when they are connected. In addition to flaws in programs [29], routes flapping, and failures, distributed systems are prone to signal fluctuations. "*We use and study the Internet, but nobody understands how or why the Internet behaves the way it does*" noted Fred B. Schneider in IEEE distributed systems online [113]. There is a poor understanding of emergent behaviors of networks.

In addition, the difference between various entities, the ways in which they communicate, the need for consistency, quality of service and reliability, taking into consideration that some parts may fail, make distributed programming a non trivial task.

Moreover, distributed algorithms are hard to modify [98]. Today's routing protocols for instance are efficient, but they are hard to change to accommodate the needs of new applications requiring improved resilience and higher throughput. In order to modify a deployed routing protocol, one need to get access to each router to modify its software.

Furthermore, one of the main difficulties of distributed programming is to ensure that the execution of distributed algorithms results in correct and efficient implementations that are faithful to the program specifications. This is particularly challenging in a distributed context, where asynchronous communication and the unannounced failure of nodes make it hard to reason about the flow of data in the related system [93].

All these difficulties are exacerbated in systems which need to deal with mobility and intermittent availability of wireless communication [72]. The dynamics of some networks, with nodes joining or leaving the networks, not to mention the various types of failures increase further the complexity

and raise considerable challenges.

Overview of existing approaches

The challenges faced in building a distributed system change depending on the requirements of the system. In particular, the diversity of components of distributed systems in terms of hardware, software, platform, etc., have led to middleware, e.g. CORBA [123], to solve the problem of heterogeneity and transparency, and to provide a common computational model.

The amount of information that needs to be stored and processed is exploding. The MapReduce framework [52, 53, 54], proposed by Google in 2004, provides a parallel programming model and associated implementations to process huge amounts of data. While Hadoop [160, 132] is designed to scale up from single servers to thousands of machines, each offering local computation and storage.

Peer-to-peer (P2P) systems are designed as well to handle high level scalability. They are widely used in various fields such as file sharing (e.g. Napster, Gnutella, Torrent), and communication networks (e.g. skype). A wide range of applications are now emerging over P2P systems, such as social networking [33, 104], multiplayer games [82, 68], mobile messaging [129], video broadcasting [92], etc.

To facilitate code reuse among systems, allow extensibility, enhance security, and provide concurrency, data-centric languages [66, 168, 169, 1, 157, 96, 94, 97, 56] have already been used. Such languages are more declarative, so facilitate programming and relieve the programmer from the intricacy of distributed programming, they parallelize well, so facilitate the execution, they manipulate explicitly data structures, so facilitate verification of their properties [56]. Declarative programming is an appealing paradigm that expresses the logic of a computation without describing its control flow [107]. Such paradigm allows to specify at a high level "what" to do, rather than "how" to do it [93]. The original vision behind this paradigm is the use of recursive query languages and processing. The declarative approach enables concise specification and deployment of distributed protocols and algorithms, and constitutes a very promising model for distributed systems [71].

Data-centric languages relying on rule-based languages, à la Datalog [22, 23, 154, 136], developed in the field of databases in the 1980's, for distributed applications, are initially proposed in UC Berkeley [98, 94], under the name declarative networking. It was shown that such languages augmented with communication primitives, allowed to express communication protocols or P2P systems with code about two orders of magnitude shorter than imperative programs, and with reasonable execution models. To our knowledge, most of the above languages follow the *forward chaining* mechanism (in the *push* mode). They are very successful in expressing various applications and programs in proactive mode, but much less to aggregation and programs in reactive mode.

Distributed systems involve a set of entities whose location and behavior may vary throughout the lifetime of the system. These constraints reflect the demand for more flexible communication models and systems, taking into consideration the dynamic nature of the related applications [60]. Content-based networking is an advanced communication model where the flow of messages is driven by the content of the messages rather than by explicit addresses assigned by source nodes [37]. Publish/subscribe systems constitute a good example of such communication model. Different variants of publish/subscribe-based schemes have been proposed such as *topic-based* [60], *content-based* [60], *type-based* [60], *location-based* [61], and *context-based* [72].

Some difficulties encountered in the publish/subscribe systems, rely in the events matching mechanism, as well as efficient routing of notifications to subscribers, while avoiding useless transmission of notifications that results in an extra level of complexity [111]. Different content-based routing ap-

proaches [37, 35, 102] have been proposed to route notifications by messages based in their content, but with the cost of increased overhead.

Our Contributions

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 Id of the node to which the message is sent.

1. Intensional Addressing

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, the destination of messages cannot be specified *a priori* by source nodes without using complex dedicated protocols. These protocols, which specify the location of data and consequently the destination of messages, may increase the complexity in terms of communication and computation. It is thus desirable to delay the evaluation of the destination of messages. Examples of such applications include:

- *Distributed hash tables*: It is a class of decentralized systems that provide a lookup service. 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 [148] or VRR [34] 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 and social networks constitute as well examples of applications where the destination cannot be specified *a priori*. In publish-subscribe systems, users publish services without specifying precise destinations to them, while subscribers express their interest to services, and receive corresponding messages, without knowledge of the publishers. In social networks, some messages can be addressed to sets of users that are out of the knowledge (e.g. when sending an advertisement) or difficult to enumerate (e.g. when sending a message to friends of a friend).

In all the above examples, target destinations are subsets of nodes. It would make things easier to have abstraction for the destination of messages, which can be cleared while traveling in the network, and evaluated by only interested nodes. Therefore, we propose a framework that offers a high-level abstraction for the destination of messages to program applications in a message-oriented manner. This framework provides a new model of messages whose destination is specified both *extensionally*, by an explicit address of a node in the network (e.g. IP, identifier), and *intensionally*, by an implicit address specified by a *selection criteria*. The latter is a set of properties declared upon application specification (application-dependent). It can be for instance a very simple property such as the type of a node (e.g. sink, sensor, etc.) or based on local data (e.g. local SQL-like query, etc.), or it can be more complex expressed by a distributed program. This framework allows messages with extensional/intensional destination, that are solved in the network while they are traveling. If

Introduction

the node associated with the extensional destination is unreachable, the intensional destination is evaluated on the fly, and a new node is identified as (extensional) destination of the message. This simplifies distributed programming, ensures persistence of data in messages, as well as resilience of the system supporting the applications.

2. Declarative language

The question now is how to represent the intensional destination selection criteria? To better answer this question, let us take an example from wireless sensor networks (WSNs). Consider an application where some sink node S fires a query R to monitor the positions of nodes which have, together with their neighbors (to avoid individual measurement errors), a temperature higher than some threshold T . How to program such queries? How to get neighbors' temperature values dynamically?

The destinations are a subset of nodes that satisfy a certain property which is based on their temperature. Suppose that each sensor node, say s_i knows its position (x_{s_i}, y_{s_i}) as well as its temperature t_{s_i} . The temperature of each neighbor is known as t_{neigh} where $neigh$ is a neighbor address. When receiving the request R from the sink S , only sensor nodes that satisfy R send their positions to the sink. In particular, for each node s_i that satisfies the following conditions sends their positions x_{s_i}, y_{s_i} to the sink S .

$$t_{s_i} > T \text{ and } [all] t_{neigh} > T$$

One of the difficulties is that the temperature of neighbors need to be fetched reactively. Therefore, a sensor node needs on-demand to send requests to get neighbors temperature and wait for *all* answers to resume the computation of the initial query.

Such example, as well as aggregation queries and reactive programs as we will see in Chapter 5, can be expressed easily using a high-level data-centric programming language, *Questlog*, that we propose in this dissertation. Questlog defines a new level of abstraction and offers features such as interaction, reactivity, autonomy, modularity, and asynchronous communication. It 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. Questlog allows to reformulate (intensional destination) queries, specify complex strategies to pull distributed data, and express efficiently aggregation queries, distributed programs and applications.

3. Programming P2P systems

To demonstrate the benefits of intensional addresses, we show that it is easy to program applications in a client-server setting, and distribute them seamlessly, under some restrictions, into P2P systems. Most of P2P applications are essentially data centric, they rely on exchange of data pushed and pulled by the peers, which can be modeled as updates over a database. In a client/server setting, the clients have views over a centralized database, they can update their views (client actions), while the server can perform updates over the whole database (system actions), *triggered* either by the server (e.g. timers) or by the client actions. We demonstrate that applications programmed as queries over a database can be distributed seamlessly, that is without changing the initial queries, from a client/server architecture, to a P2P architecture with the appropriate overlay. The distribution is done under some restrictions on the client/server applications in order to guarantee efficient distribution of data and queries relying only on the *unicast* mode. The overlay is expressed by the declarative data-centric language Netlog [66], thus resulting in a fully data centric modeling of the application.

Introduction

The overlay as well as required protocols (e.g. routing) can be chosen with respect to the applications. They can be simply modified or even changed without altering the corresponding applications. They are designed, together with the extensional/intensional destination, to guarantee smooth distribution over a network where nodes can fail. We illustrate our technique over online multiplayer games (e.g. Auction market), which has been implemented over the Netquest virtual machine [66, 29], and visualized over the QuestMonitor platform [29].

4. Data aggregation in WSN

WSNs consist of autonomous sensor nodes to monitor physical or environmental conditions, such as temperature, sound, pressure, etc., which cooperatively send their collected data to a base station, e.g. sink. To save energy and minimize the number of messages transmitted to the sink, different approaches [69, 146, 158] propose a clustering topology to first aggregate sensors collected data and then send aggregated results to the sink. We demonstrate our framework with intensional addresses on such applications. We propose a clustering protocol that decomposes the network into a set of clusters, as a set of dynamic trees, for data aggregation. Unlike classical clustering approaches, the cluster heads are virtual nodes, not known explicitly by any sensor nodes. The cluster heads are the nodes that satisfy certain selection criteria specified intensionally, which are evaluated on the fly when the (active) messages are traveling. Each node evaluates dynamically its cluster head by evaluating the intensional destination selection criteria on its local data. The protocol adapts on the fly to dynamic networks. The mobility of the code of the intensional destination selection criteria facilitates the programming of applications [156], and allows a dynamic modification of the code. We show that the proposed protocol provides high load balancing, increased persistence of data in messages as well as resilience of the system.

5. Participation in UBIQUEST ANR project

This research work is supported and financed by an ANR project UBIQUEST which involves three research laboratories CITI - INSA Lyon, LIG - Grenoble, and LIAMA - Beijing. UBIQUEST [7, 8] proposes to combine network management and data management in a single framework. Applications interact through declarative queries including declarative networking programs (e.g. routing, DHT, etc.) and/or specific data-oriented distributed algorithms (e.g. distributed join). We collaborated on the development of UBIQUEST as well as its related systems.

Organization

This dissertation is organized as follows. In Chapter 2, we present the state of the art concerning declarative programming, distributed P2P systems, and routing methodologies. We propose in Chapter 3 a framework that provides a high level abstraction of messages to allow programming in a message-oriented manner. In Chapter 4, we present an environment that distributes seamlessly client-server applications into P2P systems. We motivate and formally define in Chapter 5 the Questlog language, while in Chapter 6, we describe the implementation of the compiler that transforms Questlog rules into an executable bytecode, as well as the implementation of the system that executes the Questlog programs. To validate our framework, we present in Chapter 7 a dynamic clustering protocol that allows to aggregate data efficiently in a WSN application. In Chapter 8, we describe the Ubiquet system as well as its related visualization tool. We then conclude in Chapter 9 summarizing the overall contributions, and discussing open issues and future research directions.

State of the Art

Contents

Introduction	9
2.1 Declarative Programming	9
2.1.1 SQL-like Query Languages	10
2.1.2 Rule-based Languages	11
2.2 Distributed Algorithms	19
2.2.1 Unstructured Peer-to-Peer Systems	20
2.2.2 Structured Peer-to-Peer Systems	22
2.3 Routing Methodologies	24
2.3.1 Address-based Routing	24
2.3.2 Content-based Routing	25
Conclusion	27

2.1 Declarative Programming

Introduction

In this chapter, we summarize related work in declarative programming, distributed systems, and routing, focusing on variants of SQL as well as Datalog languages, peer-to-peer (P2P) systems, and different models of routing including address-based and content-based routing.

In Section 2.1, we survey research efforts on data-centric programming languages which constitute a very promising approach for distributed applications. Declarative query languages have already been used in the context of networks [100, 55, 31, 101]. Several systems for sensor networks, such as TinyDB [101], Cougar [55] offer the possibility to write queries in a variant of SQL. These systems provide solutions to perform energy-efficient data dissemination and query processing. 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 [66, 98, 94] and declarative overlays [96]. This approach is known as declarative networking.

In Section 2.2, we introduce the features and properties of distributed systems, and afterwards we survey a particular class of distributed systems, the P2P systems. The later are classified into unstructured systems, where objects are not placed on any particular nodes, and structured systems, where each object is placed on a specific node. We survey various approaches for unstructured (e.g. Gnutella [153, 63, 65], BitTorrent [30]) and structured (e.g. CAN [138], Chord [148], Tapestry [167]) P2P systems.

We consider a network constituted by a set of nodes that communicate by exchanging messages. A message is routed either based on its explicit destination address, such as IP-based routing protocols, or based on its content and on interests specified by source nodes, such as the publish/subscribe systems. We survey in Section 2.3 different protocols corresponding to address-based and content-based routing.

2.1 Declarative Programming

Declarative programming is an appealing paradigm that expresses the logic of a computation without describing its control flow. Such paradigm allows programmers to say "what" they want, without worrying about the details of "how" to achieve it. This is in contrast with imperative programming, in which algorithms are implemented in terms of explicit steps. Common declarative languages include those of database query languages (e.g., SQL, etc.), logic programming, functional programming, etc. In this dissertation, we are interested in declarative SQL-like query languages, as well as logic programming, such as Datalog-like languages.

The separation of a logical level, accessible to users and applications, from the physical layers constitutes the basic principle of Database Management Systems (DBMS). It is at the origin of their technological and commercial success [135]. 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, as well as to query processing techniques that optimize the declarative queries into (close to) optimal execution plans.

Another application of the declarative approach has been pursued at the network layer. The use of recursive languages, a variant of Datalog, has been initially proposed to express communication network algorithms such as routing protocols [98] and declarative overlays [96]. This approach, known as declarative networking is extremely promising. The original vision behind declarative networking is the use of recursive query languages and processing. Declarative networking [94] promotes

2.1 Declarative Programming

declarative, data-driven programming to concisely specify and implement distributed protocols and services.

2.1.1 SQL-like Query Languages

Declarative query languages have been used in the context of networks (e.g. sensor, ad hoc) for in-network query processing in order to minimize energy dissipation. Sensor networks consist of a set of autonomous nodes, which are deployed into some physical environment to monitor and collect data. Sensor nodes respond to physical signals such as heat, light, sound, pressure, etc., to produce data [32]. They operate without human interaction (e.g. route configurations, battery recharge), and present constraints such as energy, memory, and computation. The database approach to sensor networks has been motivated for two main reasons [163]. First, declarative queries are well-suited for sensor network interaction since users and application programs can issue queries without knowing how the data is generated and processed to compute the query. Second, data transmission back to a central node (e.g. sink, basestation) for offline storage, querying, and data analysis is very expensive since communication consumes high energy [59, 131]. Since sensor nodes have local computation abilities, part of the computation can be pushed into sensor nodes to aggregate or eliminate irrelevant records. Several research groups have focused on in-network query processing as a means of reducing energy consumption. Systems such as TinyDB [101], Cougar [55], and similar systems [100, 164, 31] offer the possibility to write queries in SQL-like languages. 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 [147]. We next describe briefly these systems.

TinyDB [101] is a query processing system for extracting information from a network of TinyOS sensors. It incorporates a set of features designed to minimize power consumption via acquisitional techniques. On each node, the TinyDB system addresses carefully, using the acquisitional techniques, sensors that have relevant data as well as the most convenient moment to retrieve these data. Queries are represented by a dedicated SQL-like language for query expressions definition.

Sensors monitor and collect environmental values (e.g. light, temperature). Collected data by each sensor are represented as ordered sequences of tuples under the same scheme. Tuples belong to a table which has one row per node per instant in time, with one column per attribute. The table is partitioned across all of the sensors in the network. Each sensor produces and stores its own readings. A routing tree is used to allow a base station to disseminate a query to all sensor nodes to retrieve collected data. This routing tree is formed by forwarding a routing request to all sensors in the network. Each sensor nodes chooses a parent node. This parent is responsible for forwarding the node's as well as its children's query results to the base station. TinyDB handles data streams, and includes support for aggregation which reduces the quantity of data transmitted through the network.

As TinyDB, the Tiny AGgregation (TAG) service [100] allows users to express simple declarative queries using SQL-like form. These queries are distributed and executed in networks of low-power, wireless sensors. TAG focuses on simple aggregation queries, whose execution can be distributed over an arbitrarily large set of operators. It defines both (i) aggregate operators adapted to motes with limited resources running tinyOS, and (ii) a routing strategy that imposes a routing tree onto the network: data is aggregated at every internal node in the routing tree.

Another approach Cougar [55] for in-network query processing in sensor networks has been proposed. Cougar is a distributed database system to tasking sensor networks through declarative

2.1 Declarative Programming

queries. It proposes a SQL-like language to formulate long-running queries, and source discovery techniques in order to save energy. The Cougar system forms clusters out of the sensors to allow in-network aggregation to conserve energy by reducing the amount of communication between sensor nodes.

Cougar has as input data streams generated by signal processing functions that returns output values over time. The source discovery techniques are addressed in different research directions: (i) storage point selection to minimize the number of messages to collect the data for answering a query, (ii) routing plan definition since data from different nodes must be collected in some specific sites, and finally (iii) transmission scheduling among sensor nodes such that data flows quickly from sources to storage nodes. In-network processing reduces energy consumption and improves network lifetime significantly compared to traditional centralized data extraction and analysis [163].

Yao and Gehrke [164] describe techniques to process declarative SQL-like queries over sensor networks. These techniques include in-network aggregation, implications on the routing layer, and query optimization. A query plan is used to decompose each query into flow blocks determining a set of sensor nodes that elect a leader on which a query fragment is executed. They applied this strategy to queries, complex aggregates, as well as joins.

In-network query processing is critical for reducing network traffic. It requires placing a tree of query operators such as filters and aggregations but also, as proposed in [31], correlations onto sensor nodes in order to minimize the amount of data transmitted in the network. In [31], the authors show that the problem of operator placement is a variant of the task assignment problem and they describe an adaptive and decentralized algorithm based on the neighbor exploration strategy. In particular, the placement of operators is progressively refined from neighbor node to another neighbor node until a local optimal placement is reached. More precisely, while one node is active executing an operator, a set of candidate nodes estimate the cost of running this operator. Periodically estimated costs are compared with the actual cost measured on the active node and execution is transferred to the node with the lowest cost. But the paper does not include a study on the overhead, in terms of messages exchanged, generated by the decentralized algorithm.

Most current systems support standard relational tables and SQL [5], and implement many of the performance enhancing techniques (e.g. indexing, result caching). Parallel databases achieve high performance [124]. However, they generally do not score well on the fault tolerance and ability to operate in a heterogeneous environment [5]. Abouzeid *et al.* proposes HadoopDB [5] which is a hybrid system that is designed to yield the advantages of both parallel databases and MapReduce [52, 53]. HadoopDB combines the two approaches for data analysis, achieving the performance and efficiency of parallel databases, and yielding the scalability, fault tolerance, and flexibility of MapReduce-based systems. Different approaches combine MapReduce and database systems. SQL-like languages such as HiveQL [133], Pig Latin [122], and SCOPE [41] with their related systems integrate query constructs from the database community into MapReduce-like software to allow greater data independence, code reusability, and automatic query optimization.

2.1.2 Rule-based Languages

Declarative networking is a programming methodology that enables developers to concisely specify and deploy distributed network protocols and services [95]. Declarative networking relies on the rule-based languages [22, 23, 154, 136] developed in the 1980's in the field of databases. It has been further pursued in [94], where execution techniques for Datalog are proposed. Distributed query languages thus provide new means to express complex network problems such as node discovery [17], route finding, path maintenance with quality of service [27], topology discovery, including physical

2.1 Declarative Programming

topology [26], secure networking [1], or adaptive MANET routing [91]. They have been used as well in a wide variety of areas, including distributed systems [28, 45], natural language processing [58], robotics [19], compiler analysis [86], security [77, 88, 168] and computer games [161].

In such paradigm, operations such as network management as well as applications requests and tasks are expressed as rules of the form:

$$head : - body$$

A rule is a way to express a set of premises (the *body*) and a conclusion (the *head*). There exist several strategies to evaluate a rule. There are two basic approaches for reasoning with logic programs.

- **Forward-chaining** (or Bottom-up): A form of reasoning that starts with the local data and works towards satisfying a goal. This approach evaluates rules on the local data of nodes to extract more data, until a goal is reached. An inference engine is applied. This engine searches the appropriate rules from local data store, and evaluates them in parallel. When the *body* is satisfied, the *head* is deduced. The inference engine will iterate through this process until a goal is reached.
- **Backward-chaining** (or Top-down): A form of reasoning that starts with the problem (goal) to be solved. The goal is repeatedly broken into subgoals. More precisely, this approach works from the consequent to the antecedent to check if there is data locally available that will support any of these consequents. An engine is applied. This engine searches the appropriate rules that match a desired goal, and evaluates them in parallel. If the rules are not satisfied, then the consequent is added to the list of goals and subgoals are generated.

Several languages have been proposed in the literature for high-level programming abstraction such as d3log [78], Overlog [96], NDlog [94], Snlog [45], Mozlog [108], R/Overlog [28], SeNDlog [168], Netlog [66], Dedalus [71], Webdamlog [4], etc.

We survey in the following some of the above languages, in particular Overlog [96], NDlog [94], Netlog [66], and Webdamlog [4]. They all are based on extensions to traditional rule-based language Datalog, a well-known recursive language designed for querying graph-structured data in a centralized database. These languages have interesting characteristics such as the use of negation, incremental maintenance, and/or the mobility of code. These languages are based on a relational model of data, and a fact is of the form $R(t_1, \dots, t_n)$ where R is a relation name and $t_1, \dots, t_n$ are constants. We next provide a review of Datalog, and then we describe briefly the languages and present a discussion/analysis section.

Datalog

Datalog is a centralized rule-based language. Following the conventions in Ramakrishnan and Ullman's survey [137], a Datalog program consists of a set of rules of the form:

$$p : - q_1, q_2, \dots, q_n.$$

which can be read as q_1 and q_2 and ... and q_n implies p , where p is the head of the rule, and $q_1, q_2, \dots, q_n$ is a list of literals that constitutes the body of the rule. A literal is a relation name with variables or constants as arguments. The rule can refer to each other in a cyclic fashion to express recursion. The order in which the rules are presented in a program and the order of literals in a rule body are irrelevant. The commas separating the literals in a rule are logical conjuncts (*AND*). The relations in the body and head of rules are tables. The execution of Datalog rules follows the

2.1 Declarative Programming

forward-chaining mechanism.

By convention for all languages in this dissertation, the names of literals begins with an upper-case letter, while functions, variables, and constants begin with a lower-case letter. As a Datalog example, Rules 2.1, and 2.2 form a program that defines the *transitive closure* (TC) of relation Link. The program schemas are described in Table 2.1.

$$TC(x, y) : - Link(x, y). \quad (2.1)$$

$$TC(x, z) : - Link(x, y), TC(y, z). \quad (2.2)$$

Schema	Description
Link(x,w)	Link(source, destination)
TC(x,z)	TC(source, destination)

Table 2.1: Schemas of the *transitive closure* program

The transitive closure is computed by iterating the rules over an instance of Link, that represents a given graph. Intuitively, each Datalog rule can be explained as: "if the rule body is satisfied then the rule head is deduced". For instance, for each tuple (β, γ) such that $Link(\beta, \gamma)$ holds, Rule(2.1) allows to derive $TC(\beta, \gamma)$. Similarly, for each tuple (α, β) such that $Link(\alpha, \beta)$ holds, and for each tuple (β, γ) such that $TC(\beta, \gamma)$ holds, Rule(2.2) allows to derive $TC(\alpha, \gamma)$. The rules are recursively applied till a fixpoint is obtained. In this case, the number of steps is proportional to the diameter of the graph.

We present in the following distributed languages that extend Datalog with a set of primitives in which communication between nodes in a network is enabled.

OverLog

Loo *et al.* has been proposed OverLog with a system called P2 [96] to simplify the development and the deployment of overlay networks which are used in a variety of distributed systems such as file-sharing, storage systems, and communication infrastructure. OverLog provides a high-level specification to facilitate code reuse, as well as the extension and hybridization of overlay designs. OverLog is based on an extension of Datalog. It contains constructs to specify physical distribution properties; in particular where a tuple (an entry of a table) is generated, stored, or sent, as well as continuous queries and deletion of tuples from tables.

An OverLog program is composed of tables declarations statements, and rules installed on each node of the network. The evaluation of rules follows the forward-chaining mechanism. Tables are defined explicitly via materialization statements, which specify constraints on the size and lifetime of tuple storage. For instance, the declaration:

$$materialize(Neighbor, 100, infinity, key(2)).$$

specifies that *Neighbor* is a table whose tuples are retained for 100 seconds and have infinite size. The *key(2)* construct specifies the position of the fields that form the primary key for each tuple of the table. Relations used in a program not declared as tables via materialization are treated by the P2 system [96] as streams of tuples.

OverLog defines only explicit primitive for *location specifier*. We next present the location specifier

2.1 Declarative Programming

primitive and the functionality of the language through a *ping-pong* example which calculates the latency between two nodes. In Rule (2.3), node x sends a *ping* event at time t to node y , which replies with a *pong* event, Rule (2.4). When receiving the *pong* event in Rule (2.5), node x calculates the latency which is the difference between current time and the time t .

$$Ping@y(y, x, e, t) : - PingEvent@x(x, y, e, _), t := f_now@x(). \quad (2.3)$$

$$Pong@x(x, y, e, t) : - Ping@y(y, x, e, t). \quad (2.4)$$

$$Latency@x(x, y, t) : - Pong@x(x, y, e, t_1), t := f_now@x() - t_1. \quad (2.5)$$

Schema	Description
Ping(x,y,e,t)	Ping(source, destination, eventName, currentTime)
Pong(x,y,e,t)	Pong(source, destination, eventName, time)
Latency(x,y,t)	Latency(source, destination, latency)

Table 2.2: Schemas of the *OverLog ping-pong* program

The location specifier determines the node at which the tuples in question are stored (@ in the body of a rule) or should be stored (@ in the head of a rule). For instance, the streams of tuples of the table *PingEvent* in the body of Rule (2.3) exist on node x . However, deduced head of the same rule should be stored at node y , resulting in an implicit communication between nodes.

OverLog supports function calls as well as negation and deletion. The function calls are used in the body of a rule and they are prepended by "f_". The negation is used in the body of a rule, which makes use of a keyword "not" that prepends a table. The deletion is used in the head of a rule, which makes use of keyword "delete" that prepends a table; all exact match tuples will be deleted. In addition, OverLog supports aggregation that prepends an attribute in the head of a rule, as well as random function calls used in the body, resulting in a non-deterministic language.

OverLog is used with the P2 system [96] which compiles the declarative specification of the overlay into a dataflow program. After that the P2 system executes the dataflow program to construct and maintain the overlay network.

One of the fundamental characteristics of OverLog is that the communication is implicit. There are no explicit communication primitives. All communication is implicitly generated during rule execution as a result of data placement.

NDlog

Network Datalog (NDlog) [94] has been proposed as well by Loo *et al.* to simplify the process of specifying and implementing a network design in a concise, secure and efficient manner. NDlog extends Datalog with a storage primitive, as well as aggregation and some function calls including arithmetic computations and simple list manipulation. NDlog gives the programmer explicit control of data placement and movement by using a special data type, *address*, to specify a network location. We next present the primitives of the language through an example. The following NDlog program computes the shortest paths between all pairs of nodes in a network. In Table 2.3, the attribute *pathVector* of the table *path* is a string encoding the full path, while attribute *nextHop* indicates for each path the next hop to route a data in the network.

2.1 Declarative Programming

$$\begin{aligned} Path(@s, @d, d, p, c) : - \#Link(@s, @d, c), \\ p = f_concatPath(Link(@s, @d, c), nil). \end{aligned} \quad (2.6)$$

$$\begin{aligned} Path(@s, @d, @z, p, c) : - \#Link(@s, @z, c_1), Path(@z, @d, @z_2, p_2, c_2), \\ c = c_1 + c_2, p = f_concatPath(Link(@s, @z, c_1), p_2). \end{aligned} \quad (2.7)$$

$$SpCost(@s, @d, min < c >) : - Path(@s, @d, @z, p, c). \quad (2.8)$$

$$Path(@s, @d, p, c) : - SpCost(@s, @d, c), Path(@s, @d, @z, p, c). \quad (2.9)$$

Schema	Description
Link(s,d,c)	Link(source, destination, cost)
Path(s,d,d,p,c)	Path(source, destination, nextHop, pathVector, cost)
SpCost(s,d,c)	SpCost(source, destination, cost)

Table 2.3: Schemas of the *NDlog shortest-path* program

Names of addresses, which can be variables or constants, are prepended with "@". The first attribute of each relation is the location specifier. It indicates the network storage location of existing or generated tuples. As OverLog, NDlog do not use explicit communication primitives, and all communication is implicitly generated as a result of data placement. In Rule (2.7) for example, the *path* and *#link* tables have different location specifiers. Therefore, in order to execute the rule body, link and path tuples have to be shipped in the network. The movement of these tuples will generate the messages for the resulting network protocol.

In NDlog, two types of rules have been distinguished: (i) local rules, which have the same location specifier, and (ii) non-local rules, which have different location specifiers. For instance, Rules (2.6), (2.8) and (2.9) are local, while Rule (2.7) is not local.

The evaluation of a NDlog rule depends only on communication along the physical links, so physical links are prepended by the operator "#", and are called link literal. Non-local rules are link-restricted by some link relation which represents the connectivity information of the network. Note that a link-restricted is either a local rule, or a rule with the following properties: (i) there is exactly one link literal in the body, and (ii) all others literals have their location specifier set to either the first or second attribute of the link literal.

NDlog supports different function calls as well as deletion. As OverLog, the function calls are used in the body of a rule and they are prepended by "f_". The deletion makes use of keyword "*delete*" that prepends a table in the head of a rule; all exact match tuples will be deleted. In addition, NDlog supports aggregation used in the head of a rule, as well as random function used in the body, resulting in a non-deterministic language. Unlike OverLog, NDlog does not support negation.

As OverLog, a NDlog program consists of a set of rules, which is installed on each node of a network. Its execution follows the forward-chaining mechanism. The NDlog programs are evaluated by the P2 system as well. Relations used in a rule should be declared via materialized statements. Otherwise, tuples of such relations will be treated as streams.

2.1 Declarative Programming

Netlog

Grumbach and Wang have been proposed a rule-based language Netlog [67] that allows to express distributed applications such as communication protocols or P2P applications. Netlog extends Datalog with storage and communication primitives, as well as aggregation and non-deterministic constructs.

A Netlog program is composed of a set of rules. The evaluation of rules follows the forward-chaining mechanism. Netlog programs are installed on each node of a network, where they run concurrently. Netlog defines explicit primitives for storage and communication as well as location instruction and destination specifier. We next present the primitives of the language through an example of routing. Rule (2.10) and (2.11) show a Netlog program that computes the next hop y on the path from a source s to a destination d . The affectation operator in front of the rules in the head 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, while $\uparrow\downarrow$ to both store and push them to neighbors.

$$\uparrow\downarrow \text{Route}(s, d, d) : - \text{Link}(@s, d). \quad (2.10)$$

$$\uparrow\downarrow \text{Route}(s, d, y) : - \text{Link}(@s, y), \text{Route}(y, d, z). \quad (2.11)$$

Schema	Description
Link(x,y)	Link(source, destination)
Route(x,y,z)	Route(source, destination, nextHop)

Table 2.4: Schemas of the *Netlog routing* program

The @ operator that prepends an attribute in the body of a rule is a *location instruction*. It specifies where the computation is taking place. For instance in Rule (2.10), the computation is at node s . With Netlog, deduced facts can be *unicasted* to a precise node. In this case, an @ operator prepends an attribute in the head of a rule. It is known as a *destination specifier*. In a wireless sensor network application for instance, collected data d by a sensor node x (@ in the body) is sent in unicast mode to the sink s (@ in the head), as shown in Rule (2.12).

$$\uparrow \text{Collect}(@s, d) : - \text{Sink}(s), \text{SensedData}(@x, d). \quad (2.12)$$

In addition, Netlog supports arithmetic operations used with an assignment literal ($:=$), non-deterministic constructs such as the *random choice* function, (local) *negation* as well as *deletion*. In case of plurality (e.g. various routes), one choice (e.g. route) can be chosen non-deterministically using the operator, $\diamond$, that prepends an attribute in the head of a rule. The negation operator ($\neg$) prepends a literal in the body of a rule. For instance, $\neg \text{Route}(s, d, _)$ means that there is no route from source s to destination d for any value of the next hop (underscore means "any value"). The consumption operator (!) prepends a literal (e.g. !Route(a,b,c)) in the body of a rule, and the related fact (e.g. Route(a,b,c)) is deleted after evaluation of the rule.

One of the fundamental characteristics of Netlog is that the execution of programs is local. A node cannot access the data store of another node neither for write nor for read instructions. Unlike

2.1 Declarative Programming

OverLog and NDlog, a formal semantics of Netlog has been defined, which takes into account the in-node behavior as well as the communication between nodes.

Webdamlog

Webdamlog [4, 3] has been designed by Abiteboul *et al.* to specify notably distributed web applications. It is tailored to facilitate the specification of data exchange between nodes. The Webdamlog model for distributed data management combines deductive and active rules. A node can thus exchange tuples as well as rules. Unlike previous languages, the Webdamlog language is distinguished by the mobility of rules that can be installed and executed at another node.

The model differentiates between extensional and intensional relations. Extensional relations are defined by a set of facts, where intensional relations are defined by rules as seen in Rule (2.13). Name of a relation atom, $m@p$, is composed of a relation name m and some node p separated by the symbol @. In this model, two types of rules are defined. A rule is deductive if the head relation is intensional, and otherwise the rule is active. We next present the primitives of the language through an example of application where a peer sends greetings birthday messages to friends.

$$\begin{aligned} & \text{intensional } \text{Birthday@MyIphone}(\text{string}, \text{relation}, \text{node}, \text{date}) \\ \text{Birthday@MyIphone}(\$n, \$m, \$p, \$d) : & - \text{Birthdates@MyIphone}(\$n, \$d), \\ & \text{Contact@MyIphone}(\$n, \$m, \$p). \end{aligned} \quad (2.13)$$

$$\begin{aligned} \$message@\$node(\$name, \text{"Happy birthday"}) : & - \text{Today@MyIphone}(\$d), \\ & \text{Birthday@MyIphone}(\$name, \$message, \$node, \$d). \end{aligned} \quad (2.14)$$

Schema	Description
Contact@MyIphone(\$n,\$m,\$p)	Contact@MyIphone(name, message, date)
Birthdates@MyIphone(\$n,\$d)	Birthdates@MyIphone(name, date)
Birthday@MyIphone(\$n,\$m,\$p,\$d)	Birthday@MyIphone(name, message, node, date)
Today@MyIphone(\$d)	Today@MyIphone(date)

Table 2.5: Schemas of the *Webdamlog greetings* program

In Webdamlog, an identifier prepended by the symbol \$ denotes a variable. A relation may include in its attributes a relation name as well as a node identifier. In Rule (2.14) for instance, the name of the relation in the head is a variable based on the attributes of the relation *Birthdat@MyIphone* in the body of the same rule. Consider the following fact is deduced after evaluation of Rule (2.13).

$$\text{Birthday@MyIphone}(\text{"Alice"}, \text{sendmail}, \text{inria.fr}, 02/12)$$

If the body of Rule (2.14) is satisfied, then the following deduced deduced fact will be sent to *inria.fr*.

$$\text{sendmail@inria.fr}(\text{Alice}, \text{"Happy birthday"})$$

Webdamlog supports *negation* as well as *deletion*. By default any processed fact is deleted. Because of their asynchronous nature, distributed applications in Webdamlog are nondeterministic in general. In contrast to previous languages, a program starts at a node which delegates some rules

2.1 Declarative Programming

to other nodes.

Discussion

In Table 2.6, we recapitulate the main characteristics of the surveyed languages. The recurrent theme is to provide high-level programming abstractions for dealing with distributed computation and communication. The expressivity of the languages depends on their primitives, such as for instance the *disjunction* operator offered by OverLog and supported by the P2 system. This allows to better express programs, as well as to reduce the code size. Nevertheless, as shown in particular in [118], its semantics has not been formally defined and suffers from severe ambiguities. This applies as well to the NDlog language, which is also supported by the P2 system. However, the NDlog language and the P2 system have gained increasing interest in the systems and networking community. The main applications are network protocols, including overlays, mobile ad hoc networks, and distributed hash tables [96, 98, 94, 93, 95, 90]. One of their important property is the incremental maintenance, especially for cascaded deletions, resulting in an update of the network state. They have been increasingly adopted by systems and networking researchers [168, 90].

	Overlog	NDlog	Netlog	Webdamlog
Negation	Yes	No	Yes	Yes
Disjunction	Yes	No	No	No
Semantics	No	No	Yes	Yes
Mobile code	No	No	No	Yes
Cascaded deletions	Yes	Yes	No	Yes
Mode	push	push	push	push
Bytecode	dataflow	dataflow	SQL	dataflow
System	P2	P2	Netquest	WebdamLog

Table 2.6: Summary of languages characteristics (primitives, properties and system)

In contrast to OverLog and NDlog, the semantics of Netlog has been formally defined [67]. In particular, Netlog admits a well-defined distributed fixpoint semantics, and bounds on the complexity of the distributed execution. An important characteristic of Netlog is that the execution of Netlog programs is local. This simplifies the semantics of (local) negation. It also facilitates the design of protocols, as well as the verification of programs [56].

Likewise, the semantics of Wedamlog has been formally validated [4]. In addition, Webdamlog is distinguished by the notion of delegation that allows a peer to install rules at other peers. This is in contrast to all other languages which are based on a distribution of the program before the execution. Because of delegation, the Webdamlog language is particularly well suited for distributed applications, providing support for reactions to changes in evolving environments. As shown in [4], the power of delegation critically depends on the exact definition of the language, obtained by allowing or restricting delegations.

Other important proposals [18, 71, 75] extend Datalog with time and space. Hellerstein *et al.* has been proposed Dedalus [18, 71] which extends Datalog, and adds an integer *timestamp* field to every tuple. The key idea is the use of time as an organizing principle for distributed systems. The *monotonic* property offered by Dedalus facilitates its semantics. While Interlandi, Tanca, and Bergamaschi [75] have introduced a semantics of a distributed version of Datalog⁺ specifically tailored for distributed programming in synchronous settings.

2.2 Distributed Algorithms

One can notice that all surveyed languages use the forward-chaining mechanism, which follows the *push* mode. When receiving a fact, the appropriate rules are evaluated, and deduced facts are either stored locally or sent to other nodes. These languages are very successful in expressing various applications and protocols in proactive mode, but much less in reactive mode. For instance, suppose as a particular example a node fires a query in order to aggregate a temperature in a tree constructed over a network. Each parent node after firing the (sub)query, waits for all children temperatures to resume the evaluation of the (sub)query. Of course, this can be expressed with the above languages, but much less easily than in a language that supports the *pull* mode (Backward-chaining), as well as with some complexity in code (e.g. much more rules are required). In this dissertation, we propose a new language *Questlog* (Chapter 5) that follows the *pull* mode. It allows to express efficiently reactive programs as well as on-demand application queries. Unlike the *push* mode, the *pull* mode requires bookKeeping pending queries, resulting in a memory usage at each node to match answers with their pended queries.

2.2 Distributed Algorithms

Distributed systems and applications have gained much interest in recent years. The development of wide-area distributed applications has led to growing demands for fundamental properties. We next present the main properties to get ideal distributed systems [49].

- *Heterogeneity*: As a distributed system evolves it tends to grow more diverse [144]. Various entities in the distributed system must be able to interoperate with one another, despite differences in hardware architectures, operating systems, communication protocols and networks, programming languages, software interfaces, security models, data formats, and system managers.
- *Openness*: It is the possibility to extend the system. This is determined primarily by the degree to which new services can be added and be made available for use [49]. Interfaces should be cleanly separated and publicly available to enable easy extensions to existing components and add new components. The challenge is to tackle the complexity of distributed systems consisting of many components engineered by different people.
- *Security*: Access to resources need to be secured to ensure that only known users are able to perform allowed operations. Security for resources has three components [49]: confidentiality (protection against unauthorized individuals), integrity (protection against corruption), and availability (protection against interference). Other challenges such as denial of service attacks, and security of mobile code traveling across a network need to be taken into account.
- *Scalability*: Distributed systems must remain effective when there is a significant increase in the number of resources and users [49, 151, 119]. Scalability has 3 dimensions [151, 119]: (i) *size*; number of users and resources to be processed, (ii) *geography*; users and resources may lie far apart, and (iii) *administration*; related system spans many independent administrative organizations. These dimensions affect the reliability, performance, and administrative complexity of distributed systems.
- *Failure handling*: Distributed systems involve a set of components (hardware, software, communication). When faults occur, they may produce incorrect results or may stop before they have completed the intended computation. Failures in a distributed system might be partial or even total. Some components may fail while others continue to function. Some failures can be detected, and sometimes it is difficult or even impossible to detect other failures, such as a remote crashed server in the Internet [49].

2.2 Distributed Algorithms

- *Concurrency*: Resources can be shared by many users, which may attempt to access a shared resource at the same time. For example, a data structure that records bids for an auction may be accessed very frequently when it gets close to the deadline time. The process that manages a shared resource could take one user request at a time. But that approach limits throughput. Therefore multiple user requests need to be handled and processed concurrently.
- *Transparency*: Distributed systems should be perceived as a single unit for users rather than as a collection of autonomous systems, which are cooperating [49, 151, 116]. The users should be unaware of where the services are located, as well as users requests from a local machine to a remote one should also be transparent. The concept of transparency can be applied to several aspects which involve access, location, replication, migration, concurrency, failure, and performance transparency.

Different types of distributed systems exist nowadays, including cluster, grid, P2P, and pervasive systems. In this section we are interested in the P2P systems where all nodes (peers) share equivalent responsibility for processing data, and act at the same time as clients and servers. Before proceeding to introduce the functionality of the P2P systems, our interest in such class of distributed systems is due to the possibility to handle a high volume of traffic by distributing the load across many peers. Because they do not rely exclusively on central servers, they scale better and are more resilient than traditional client/server systems in case of failures or traffic bottlenecks. In Chapter 4 we propose an environment that distributes seamlessly, under certain restrictions, client/server applications into P2P systems with the appropriate overlays. The communication relies on messages with extensional/intensional addresses (Chapter 3), which can be evaluated on the fly, and ensure persistence of data in messages.

Let us now continue to introduce the functionality of P2P systems. A common use of such systems is as an object store [87]. In its role as a client, a peer can create objects that are stored in the system and inject queries that find objects with certain properties. In its role as a server, a peer provides storage capacity for objects and answers queries for objects stored locally. If the P2P network is unstructured, objects are not placed on any particular nodes. Queries are forwarded to all nodes, and each node receiving such a request checks whether its local objects fulfill the query. In comparison, in a structured network each object is placed on a specific node (or subset of nodes), typically the nodes whose hashed node ids are closest to the hashed value of one or more attributes of the object. Queries with search keys on these attributes can then be easily routed to the nodes that contain matching objects. We next survey various approaches for unstructured and structured P2P systems.

2.2.1 Unstructured Peer-to-Peer Systems

The unstructured P2P centralized model was first popularized by Napster [117], which uses a centralized server to locate content, resulting in scalability limitations. Subsequent P2P systems adopted decentralized search algorithms. Gnutella [153] is a decentralized protocol for distributed search and file sharing on a flat topology of peers. Gnutella does not have any precise control over network topology or file placement. Peers provide interfaces to let users fire queries and show results [99]. At the same time, they process queries received from other peers. In particular, they check for matches against their local data, and respond with applicable results. To join the system, a new node first connects to one of several known nodes that are always available. To locate a data item, a peer broadcasts queries in the network with a certain radius according to a specified time-to-live (TTL). Such design is resilient to peers entering or leaving the system. However, the search mechanism is not scalable and generates unexpected loads on the network [99].

2.2 Distributed Algorithms

Enhanced versions of Gnutella have been proposed in [63, 65] to improve routing performance. They adopt the concept of super-peers or ultra-peers (high capacity peers) that act as proxies for lower capacity peers. A leaf peer selects a number of ultra-peers and sends them its file list. Ultra-peers perform query processing on behalf of their leaf peers. A query of a leaf peer is sent to its ultra-peers which flood the query to all ultra-peer neighbors. At the same time, ultra-peers perform matching when receiving queries from other peers. This reduces the lookup traffic at the leaves. The new versions of Gnutella have better performance but remain limited since they still use the flooding mechanism across ultra-peers.

The Gnutella design has been modified and improved by a new system Gia [43] that dynamically adapts the overlay topology and the search algorithms in order to accommodate the natural heterogeneity present in most P2P systems. Gia replaces Gnutella's flooding with random walks, and introduces a flow control algorithm, dynamic topology adaptation as well as one-hop replication.

Another search mechanism is proposed in [84] where the authors provide a cluster-based architecture for P2P systems (CAP). This system employs network-aware clustering technique [83] for content location and routing. CAP uses a centralized server to perform network-aware clustering and cluster registration. To help query lookup and forwarding, each cluster has some delegate peers that act as directory servers for objects stored at peers within the same cluster. CAP does not guarantee that an existing object will be found.

In contrast to Gnutella, BitTorrent [30] is a centralized P2P system that uses a central location (tracker) to manage users' downloads and coordinate file distribution. Rather than downloading a file from a single source server, the BitTorrent related protocol allows users to join a set of hosts to download and upload from each other simultaneously. The most notable feature of BitTorrent is its use of *tit-for-tat* trading to incentivize users to give each other higher bandwidth service [48].

To share a file, a peer first creates a *torrent* file, which is a descriptor file that contains metadata about the file to be shared such as its length, name, hashing information, and the URL of a tracker. After that the torrent file is typically published on an ordinary web server, and registered on the tracker. The tracker keeps track of all the peers who have the file and lookup peers to connect with one another for downloading and uploading.

BitTorrent decomposes files into pieces of fixed size, which are included in the torrent file. To download a file, a peer first obtains corresponding torrent file from the web server and connects to the specified tracker, which responds with a random list of contact information about the peers which are downloading the same file. Downloaders then use this information to connect to each other and exchange various pieces. When a peer finishes downloading a piece and checks that the hash matches, it announces that it has that piece to all of its peers. A downloader which has the complete file, known as a *seed*, must send out at least one complete copy of the original file [99].

In most of previous unstructured systems, search used simple forwarding mechanisms such as flooding or random walks were often inefficient or unreliable [87]. Modern unstructured overlays like BubbleStorm [152] or the similar approach in [62] provide reliable and exhaustive search even in very large networks. In [62], the authors present a technique for object location. A peer installs object references at a set of randomly selected peers. A query to the object is routed to another set of random peers selected independently of the installation procedure. The high probability of a non-empty intersection between these two sets forms the basis for the search mechanism. The BubbleStorm [152] approach is a probabilistic search system, which probabilistically guarantees that the application's query evaluator runs on a computer containing the sought data. This system uses a large number of replicas for each object placed randomly in the overlay to enable their search algorithms.

2.2 Distributed Algorithms

2.2.2 Structured Peer-to-Peer Systems

Structured P2P systems provide efficient insertion and retrieval of content in a large distributed storage infrastructure using indexing mechanism. In effect, Content Addressable Network (CAN) [138] is a distributed P2P infrastructure that provides hash table-like functionality on Internet-like scales. It organizes an overlay into a d -dimensional Cartesian coordinate space on a d -torus. The coordinate space is dynamically partitioned among all the nodes in the system to store (key,value) pairs. Each node owns its distinct zone within the overall space. CAN uses a uniform hash function that maps keys into points in the coordinate space. For instance, to store a pair (K_i, V_i) , key K_i is deterministically mapped onto a point P_j by the hash function. The corresponding (key,value) pair is then stored at the node that owns the zone within which the point P_j lies. To retrieve an entry corresponding to key K_i , a node applies the same deterministic hash function to map K_i onto point P_j and then retrieves the corresponding value from the point P_j . Note that if the point P_j is not owned by the requesting node or its immediate neighbors, the request is routed through the CAN infrastructure until it reaches the node in whose zone P_j lies. Each node maintains a routing table with its adjacent immediate neighbors. A node routes a message towards its destination by simple greedy forwarding to the neighbor with coordinates closest to the destination coordinates. When a node receives a join request, it splits its zone in half and assigns one half to the new node. Failure of a node is handled by a takeover algorithm. There are open research questions on CAN's resiliency, load balancing, locality and latency [99].

Another structured P2P approach for lookup service such as Chord [148] has been proposed. Chord maps keys onto nodes. It uses consistent hashing [80] that allows to balance load and to let nodes enter and leave the network with minimal interruption. The consistent hash function assigns each node and data key an m -bit identifier using the hash function SHA-1. A node's identifier is chosen by hashing the node's IP address, while a key identifier is produced by hashing the data key. Identifiers are ordered in an identifier circle modulo 2^m , called Chord *ring*. A key is assigned to the first node whose identifier is equal to or follows the identifier of k in the identifier space. This node is called the successor node of key k .

Lookup queries pass around the ring via successor pointers until getting responsible node. A portion of the Chord protocol maintains these successor pointers, thus ensuring that all lookups are resolved correctly. However, this resolution scheme is inefficient: it may require traversing all nodes to find the appropriate mapping. To accelerate this process, Chord maintains additional information. Each node maintains a routing table with m entries, called the *finger* table. A finger table entry includes both the Chord identifier and the IP address of the relevant responsible node. In this way, the lookup queries are executed more efficiently. When a node joins the system, the successor pointers of some peers need to be updated. Chord uses a stabilization protocol to update the successor pointers and the finger table. At the same time, some of the keys that are assigned to the successor of the new node α , must be assigned now to the node α . Similarly, when a node fails, all its keys are assigned to its successor. It is possible that a node does not know its new successor. To avoid such situation, each node maintains a *successor list* which contains its r nearest successors. If node n notices that its successor has failed, it replaces it with the first live entry in its successor list.

In Chord, the ring is maintained using periodic stabilization procedure, resulting in a high communication costs. In [89], the authors investigate the per-node network bandwidth consumed by maintenance protocols in P2P networks. They prove that an appropriately modified version of Chord's maintenance rate is within a logarithmic factor of the optimum rate. Alima *et al.* [16] proposes a mechanism in their Distributed K -ary Search (DKS), which is similar to Chord, to reduce

2.2 Distributed Algorithms

the communication costs incurred by Chord's stabilization procedure. Unlike Chord, DKS uses a *correction-on-use* technique to correct on-the-fly expired routing entries while performing lookups and key/value insertions.

Let us now review Tapestry [167] which is a P2P overlay network that uses a variant of the Plaxton *et al.* [130] distributed search technique as well as additional mechanisms to provide availability, scalability, and adaptation in the presence of failures and attacks. Plaxton *et al.* [130] proposes a randomized algorithm which maintains and locates the addresses of copies of objects which are connected to one root node. Tapestry uses multiple roots for each data object, resulting in more dynamics and fault tolerance.

Nodes that participate in the overlay are assigned *nodeIDs* uniformly at random from a large identifier space. Tapestry maps an identifier to a unique responsible live node. To deliver messages, each node maintains a routing table to neighbors. Responsible nodes only store pointers. When routing toward a root, a message is forwarded to neighbor whose *nodeID* is progressively closer to the root. Each node along the route stores a pointer mapping the object. Multiple nodes can publish pointers to the same object. This happens when copies of the object are created at different nodes, which publish messages and create location pointers on the way. Tapestry provides adaptation to faults and evolutionary changes, while providing eventual recovery from problems.

Maintaining a running DHT requires non-trivial operational effort. A shared deployment could amortize this operational effort across many different applications. Rhea *et al.* proposes OpenDHT [142], a shared public DHT service. It operates on a set of infrastructure nodes. Clients run application code that invokes the OpenDHT service using remote procedure call (RPC). Each node participates in the DHT's routing and storage, and at the same time acts as a gateway through which it accepts RPCs from clients. OpenDHT uses the storage model interface with its simple put/get operations, as well as a client library, recursive distributed rendezvous (ReDiR), which provides the equivalent of a lookup interface for any arbitrary set of machines. In OpenDHT, a set of functions is available allowing mainly to insert, remove, change, and get values, as well as other operations [142]. To find a service node, a client performs a lookup, which takes a key chosen from the identifier space and returns the node whose identifier most immediately follows the key. OpenDHT uses a storage allocation algorithm to provide fairly storage between clients and with high utilization, and avoid long periods in which no space is available for new storage requests. The OpenDHT suffers from high latency. In [141], the authors highlight the problem of OpenDHT slow nodes, and show that their effect on overall system performance can be mitigated through a combination of delay-aware algorithms and a moderate amount of redundancy.

To end with structured P2P systems, let us see Virtual Ring Routing (VRR) [34] which provides both traditional point-to-point routing and DHT routing to the node responsible for a hash table key. It is implemented directly on top of the link layer. VRR never floods the network and uses only location independent identifiers to route. As Chord [148], nodes are organized into a virtual ring ordered by their identifiers and each node maintains a small number of routing paths to its neighbors in the ring. The nodes along a path store the next hop towards each path endpoint in a routing table. VRR uses these routing tables to route packets between any pair of nodes in the network: a packet is forwarded to the next hop towards the path endpoint whose identifier is numerically closest to the destination. The DHT functionality offered by VRR is particularly useful because it can be used to implement scalable network services in the absence of servers.

2.3 Routing Methodologies

Nodes in a network communicate by exchanging messages which are addressed to endpoint references (destinations). An endpoint reference's role is to identify the node that will eventually deal with the content of the message. Sometimes the specified destination is no longer valid due either to failure or to mobility of nodes. It is also possible that the destination no longer deals with messages of that particular type. This is where content-based routing comes in. In content-based routing, a message is routed by being opened and then having a set of rules applied to its content. These rules are used to specify which parties are interested in it, allowing to determine the destination where it should be sent.

We propose in Chapter 3 a new model of messages whose destination combines the best features from both methodologies (address-based and content-based routing) including efficiency, flexibility, reliability, simplicity, and quality of service. Let us in this section overview various protocols of each methodology.

2.3.1 Address-based Routing

In address-based routing, nodes are identified by their addresses (e.g. IP, Id), and messages are routed based on their explicit addresses. When a source node has data to send to a destination node, the data is routed through multihop routing if the source is not directly connected to the destination. According to a routing table, a node forwards the data to the nexthop on the path to the destination. There are different types of protocols (e.g. RIP [103], OSPF [115], EIGRP [150], BGP [139]) that are used to determine the best route to send data to precise destination(s) over a network. As well, in ad hoc networks, several routing protocols have been proposed such as DSDV [128], AODV [125], and OLSR [47]. We next present an overview of some of these protocols.

Routing Information Protocol (RIP) [103] is a distance-vector routing protocol based on the Bellman-Ford algorithm which has been used for routing computations in networks. RIP employs the hop count as a routing metric, and prevents routing loops by implementing a limit on the number of hops (maximum 15) allowed in a path from a source to a destination. In RIP, each node (router) periodically sends a copy of its entire routing table to all neighbors. Those nodes will then update their tables with that information, then send their entire routing table to their neighbors. This cycle will continue until all of the nodes in the network have exchanged information about each other. This design is inappropriate for larger networks due to the limitations of the size of networks that RIP can support.

Open Shortest Path First (OSPF) [115] is a link-state routing protocol. Each node (router) maintains an identical database describing the topology. Each individual piece of this database is a particular node's local state. The node distributes its local state by flooding. From this database, a routing table is calculated by constructing a shortest-path tree. Unlike RIP, OSPF recalculates routes under topological changes detection, utilizing a minimum of routing protocol traffic. OSPF provides support for equal-cost multipath. An area routing capability is provided, enabling a reduction in routing protocol traffic. In addition, all routing exchanges are authenticated. The OSPF protocol requires increased amount of memory and extra processing.

In contrast to RIP and OSPF, we next review protocols designed for MANETs. Destination-Sequenced Distance Vector (DSDV) [128] is a table-driven protocol which uses the Bellman-Ford algorithm to calculate routes. DSDV maintains a routing table with entries for all nodes in a network. Routing information is propagated through periodic update mechanisms used by DSDV. To prevent loops, each route table entry is tagged with a sequence number so that nodes can quickly

2.3 Routing Methodologies

distinguish stale routes from the new ones and thus avoid formation of routing loops. The sequence number from each node is independently chosen but it must be incremented each time a periodic update is made. In DSDV, the cost metric used is the hop count, which is the number of hops it takes for the packet to reach its destination. DSDV is not suitable for highly dynamic networks, and costly in terms of energy due to periodic update of routing tables.

Optimized Link State Routing (OLSR) [47] is a table-driven protocol which uses HELLO and topology control (TC) messages to discover and broadcast link-state information throughout the network regularly. Nodes receiving this topology information compute next hop destinations for all nodes. Each node selects a set of its neighbor nodes as multipoint relays (MPR). In OLSR, only MPRs are responsible for transmitting broadcast messages and constructing link state. OLSR floods topology data frequently enough over the network to make sure all nodes are synchronized with link-state information. In OLSR, the periodic topology information update results in a high bandwidth usage.

Unlike previous protocols, Ad hoc On-Demand Distance Vector (AODV) [125] operates reactively to find routes only on-demand. Reactive routing protocols offer quick adaptation to dynamic link conditions, low processing and memory overhead, and low network utilization. In AODV, when a route to a given destination does not exist, a route request (RREQ) message is flooded. Each node receiving the request caches a route back to the originator of the request. Once the RREQ message reaches the destination or an intermediate node that has a fresh enough route to the destination, the node responds by unicasting a route reply (RREP) message back to the originator along the reverse path. Nodes that receive a RREP set up forwarding entries in their routing tables, pointing to the node from which they received RREP message. AODV uses destination sequence numbers to ensure loop freedom, avoiding counting to infinity problem associated with classical distance vector protocols such as RIP [103]. In AODV, multiple RREP messages in response to a single RREQ can lead to heavy control overhead.

2.3.2 Content-based Routing

Unlike address-based routing, content-based routing adopts a new style of communication where messages do not carry any explicit address. They are routed based on their content and on interests specified by nodes [51]. We next survey different approaches and protocols that follow such communication model.

Directed diffusion [73, 74] is a data-centric paradigm for data dissemination in sensor networks. Data generated by sensor nodes is named by attribute-value pairs. A source node (sink) diffuses a request, which is transformed into an interest, towards nodes in a specified region. This dissemination sets up gradients within the network. Data matching the interest is then sent back towards the source node, along the reverse path of interest propagation. Intermediate nodes as well might aggregate the data. Directed diffusion is energy efficient since it is on-demand and there is no need for maintaining global network topology [15]. However, it is not suited for applications that require continuous data delivery to the sink. Directed diffusion [73] builds a single-path routing. To route around failed nodes, it assumes periodic flooding of events that enable local re-routing around failed nodes, resulting in a high energy consumption. Ganesan *et al.* [64] suggests employing multiple paths in advance to save energy and to increase resilience to node failure. When a node on the primary path fails, data can go on an alternate path.

As a generalization of directed diffusion [73], Constrained anisotropic diffusion routing (CADR) [46] proposes two techniques (i) information-driven sensor querying (IDSQ) to optimize sensor selection, and (ii) constrained anisotropic diffusion routing to direct data routing and incrementally

2.3 Routing Methodologies

combine sensor measurements so as to minimize an overall cost function. The idea is to query sensors and route data in a network in order to maximize the information gain, while minimizing the latency and bandwidth. This is achieved by activating only the sensors that are close to a particular event and dynamically adjusting data routes. The major difference from directed diffusion [73] is the consideration of information gain in addition to the communication cost. In CADR, each node evaluates an information/cost objective and routes data based on the local information/cost gradient and end-user requirements. The information utility measure is modeled using standard estimation theory.

Another approach Content Centric Networking (CCN) [76] for content-based routing has been proposed. CCN is a networking architecture that uses named content as its central abstraction. CCN has no notion of host as its lowest level. CCN communication is driven by the consumers of data. There are two CCN packet types, Interest and Data. A consumer asks for content by broadcasting its Interest to all nodes. Each node keeps track of Interests towards content sources. Any node that has data satisfies the interest respond with a Data packet towards originator requestor. To provide reliable and resilient delivery, CCN Interests that are not satisfied in some reasonable period of time must be retransmitted, resulting in some overhead and delay.

Let us move on to review publish/subscribe schemes which constitute a good example of systems with publishers who do not have to specify precise receivers (subscribers), leaving the system matching them. Publish/subscribe schemes constitute a paradigm for developing systems which enable the decoupling of interacting components, separating communication from computation. They consist of three principal components: subscribers, publishers, and a mediator. Subscribers express their interest in an event or a pattern of events. Publishers generate events. The mediator is responsible for matching events with the interests and sending them to the subscribers. Different classes of publish/subscribe systems have been proposed. *Topic-based* systems [60, 112] rely on the notion of topics, where participants publish events and subscribe to individual topics. Subscribers specify their interest by subscribing to a topic, also known as channel, subject, or group [60]. Each event produced by the publisher is labeled with a topic and sent to all the topic subscribers. This class of publish/subscribe systems is in general a static scheme with limited expressiveness. In [112], the authors propose a distributed clustering algorithm that utilizes correlations between user subscriptions to dynamically group topics together, into virtual topics (called topic-clusters), and continuously adapts the topic-clusters and the user subscriptions. However, handling carefully the topic-clusters is not an easy task.

Content-based systems [60, 36, 37, 38, 39] allow filtering on the content of an event. Subscribers specify their interest through event filters, which are boolean queries on the events content. Published events are matched against the filters and only those events that satisfy the filters are delivered to the subscribers. This approach might result in high numbers of topics and potentially redundant events that increase the overhead. Sivaharan *et al.* [145] introduces GREEN, a dynamically configurable middleware, based on component approach, to support flexible system on top of diverse network types and heterogeneous device types. It provides as well pluggable publish/subscribe interaction types such as topic-based and content-based. Content-based publish/subscribe is highly expressive, but requires sophisticated protocols that have higher runtime overhead.

Another class of the publish/subscribe approach is *type-based* systems [60] which 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 [61] support location-aware communication between participants based on positioning mechanisms. The key difference between *location-based* and previous schemes lies in the existence of an external context that impacts the matching of events and subscriptions. *Context-based* [50] systems capture the situation or context of

2.3 Routing Methodologies

information. The idea is that not only the information content of messages is relevant to determine the information flow, but also the context in which this information has been produced and its relationship with the context of subscribers.

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 [111]. Different content-based routing approaches [111] have been proposed to route efficiently notifications by messages based in their content. In [37], a routing scheme is introduced based on a combination of a traditional broadcast protocol and a content-based routing protocol. The broadcast layer handles each message, while the content based layer limits the propagation of each message to only those nodes that match the content of the message. However, this approach suffers from a high communication complexity to build spanning trees to send notifications. In [102], the authors propose a new method to provide end-to-end reliability based on the publish/subscribe system. They develop a mechanism for a message-loss detection, and a routing scheme to deliver request messages to nodes that provide repairs. This approach is costly in terms of memory due to messages caching, and in terms of communicated messages due to message-loss detection and recovery, and a re-publishing of messages to repair. In most of the platforms for topic-based event notification, the events are delivered via a supporting distributed data structure (typically a multicast tree), which need to be continuously maintained, resulting in a high overhead. Milo *et al.* [112] devises a technique for reducing this maintenance overhead. This technique dynamically groups topics together, into virtual topics, and thereby unifies their supporting structures and reduces costs.

Conclusion

During this study of the state of the art, we overviewed approaches in declarative programming, P2P systems, and routing. Content-based routing approaches such as the publish/subscribe systems [60, 37, 35, 102] propose a message-oriented communication facility based on the idea of interest-driven routing. Some difficulties and complexities encounter the publish/subscribe systems relying on the events matching mechanism, and the efficient routing of notifications to subscribers. In comparison with traditional IP-based routing, the content-based routing approach offers an abstraction for programming applications. It relieves source nodes of the need to know where the messages should be sent. But, some mapping must exist between the message's content and the nodes that ultimately consume the messages. Depending on the application, the appropriate (IP/content-based) routing model is used. However, in some applications where nodes may fail and new nodes may join, combining between the two models of routing provides interesting properties such as simplicity, efficiency, persistence and resilience. The literature, however, does not provide details on such approach. In Chapter 3, we propose a new model of communication that combines address-based and content-based routing. This model is used efficiently with the distribution of client-server applications (data and queries) into P2P systems, Chapter 4, and with the dynamic construction of clusters to aggregate data in wireless sensor networks, Chapter 7 .

As we have seen in Chapter 1, programming distributed systems is challenging. In the literature, different approaches such as [96, 94, 66, 4] propose declarative languages to allow programming distributed systems at a certain level of abstraction. However, they all follow the forward-chaining evaluation mechanism. They are very successful in expressing various programs in proactive mode, but less so in reactive mode. We propose in Chapters 5 and 6 a data-centric language well-suited to aggregation and reactive programs, and it is used as well to manipulate content-based specification represented by queries, to facilitate programming distributed systems and applications.

Extensional and Intensional Destinations

3

Contents

Introduction	29
3.1 Message Model	30
3.2 Destination Execution Priority Order	31
3.2.1 Extensional destination higher priority order	31
3.2.2 Intensional destination higher priority order	32
3.3 Intensional Destination Strategies	34
3.3.1 Message decision before processing	34
3.3.2 Message decision after processing	34
3.4 Intensional Destination Specification	35
3.4.1 Intensional destination as SQL query	35
3.4.2 Intensional destination as Questlog query	36
Conclusion	37

Introduction

In recent years, the wide development of networks especially with the use of wireless technologies raised tremendously the number of network applications in different domains. The trend towards ubiquitous wireless communications interconnecting an increasing number of heterogeneous devices (nodes) such as sensors, PDA's, wearable computers, etc, leads to an increase of the complexity and dynamics of communication networks.

Networks consists of a set of nodes that communicate by exchanging messages. Due to the mobility of nodes which may join or leave networks as well as they may fail, programming (distributed) applications is challenging.

In some distributed applications, as we have seen in Chapter 1, data are fragmented over nodes with no prior knowledge on their location. Nodes communicate by exchanging messages with a *payload*, the content of the message, and a *destination*, an explicit address to which the message is sent. Dedicated protocols that specify the location of data and consequently the destination of messages, may increase the communication and computation complexity. It is desirable to delay the evaluation of the destination of messages.

Routing by content is an appealing paradigm where the flow of messages is driven by the content of the messages rather than by explicit addresses assigned by source nodes. We propose a new model of messages whose destination is specified both *extensionally*, by an explicit address, and *intensionally*, by an implicit address specified by a *selection criteria* or *properties* declared by the application (application-dependent). This model allows to program (distributed) applications in a message-oriented manner, allowing messages with intensional destinations, that are solved in the network while they are traveling.

Having a message with both extensional and intensional destinations, two ways of execution are possible. When receiving a message, a node might first check either the extensional destination or the intensional destination. We next briefly show examples of applications with different priority order of execution.

Chord [148] is a peer-to-peer lookup protocol where nodes are organized in a virtual ring in order of increasing identifier. The virtual ring is maintained periodically. Each node in the ring, say α , is responsible for an interval of keys $[\alpha, \text{succ}(\alpha)]$, where succ is the first node that follows α . A key is the hash result of a data. Suppose node α has data to send with key k . It evaluates locally, based on local data, the responsible node which is "*the first node whose identifier is equal to or follows k* ". In this case, suppose that the responsible node is node β , then the data is sent to β in a message through multi-hop routing. Suppose now that when the message is traveling in the network, node β leaves the network. Intermediate node γ cannot route the message to β , resulting in a lost of the message.

To facilitate the programming of chord, and to increase the persistence of data in messages, we propose to abstract the destination nodes. Indeed, we use our new model of messages with destination specified both extensionally and intensionally. In particular, node α sends the data in a message with extensional destination β and intensional destination "*the first node whose identifier is equal to or follows k* ", through multi-hop routing. If node β leaves the network, then intermediate node γ cannot route to β . In this case, intermediate node γ executes on the fly the intensional destination, gets as a result a new extensional destination, and then routes the message to the new extensional destination. This ensures persistence of data in messages, as well as resilience of the execution.

Each time a message is received, a node first checks if it is the destination. In particular, it checks

3.1 Message Model

if the extensional destination is equal to its address. Thus, chord is an example of application where the extensional destination has the highest priority order.

Let us now see an example where the intensional destination has the highest priority order. Suppose a wireless sensor network application where nodes are organized in clusters. Members send their collected data to their appropriate cluster head which aggregates the data and sends it to a sink. Suppose that the cluster head is not known to a node, then when a sensor has collected data to send, it specifies based on local data a cluster head identifier, following an intensional destination *selection criteria*, and then sends to it the data. When a node receives a message, it checks if it is the cluster head, by executing the intensional destination. If it is the cluster head, it processes the message, and otherwise sends the message to the cluster head.

When receiving a message with (a specified) intensional destination, different strategies can be applied. For instance, a node can (i) process the intensional destination of the message and at the same time send the message to other nodes, or (ii) process the intensional destination of the message and then decide what to do with the message according to the results of processing. Using the second strategy, the node might decide to route the message randomly, send in multicast or unicast following the set of results, or even discard the message.

The intensional destination of a message is specified by a selection criteria. It may be for instance a very simple property based on local data of a node such as the type of a node, or it can be more complex following a distributed program. We propose to specify the intensional destination by either (i) a SQL query executed locally on a node, or (ii) a query specified in the Questlog language [12] which allows to reformulate queries and express complex strategies to pull distributed data.

The Chapter is organized as follows. In the next section, we present the message model. In Section 3.2, we present the execution priority order of extensional and intensional destinations of a message. Section 3.3 is devoted to discuss the strategies of intensional destinations, while in Section 3.4 we describe two specifications for the intensional destination selection criteria.

3.1 Message Model

A network is composed of a set of nodes that communicate by exchanging messages of the form:

$$msg = \langle Payload, Destination \rangle$$

where *Payload* is the content of the message, and *Destination* is the Id of the node to which the message is sent. Figure 3.1 shows the message model. In distributed network applications, data is fragmented over participating nodes that communicate either for querying or updating the data. Therefore, the payload can consist either of *data* or *queries*.

In an increased number of applications, nodes in general have no knowledge of the location of data. The destination in some applications cannot be specified explicitly, as seen in Chapter 1, due to a lack of knowledge of the nodes that satisfy certain properties or criteria.

We propose a new model of messages whose destination, as seen in Figure 3.1, is specified both *extensionally*, by an explicit address of a node in the network (e.g. IP, identifier), and *intensionally*, by a *selection criteria* (e.g. query). The selection criteria is application-dependent. It is a set of properties declared upon application specification. It can be for instance a very simple property based on local data of a node such as the type of a node, or it can be more complex expressed by a distributed program. The destination of messages is formally defined as follows.

3.2 Destination Execution Priority Order

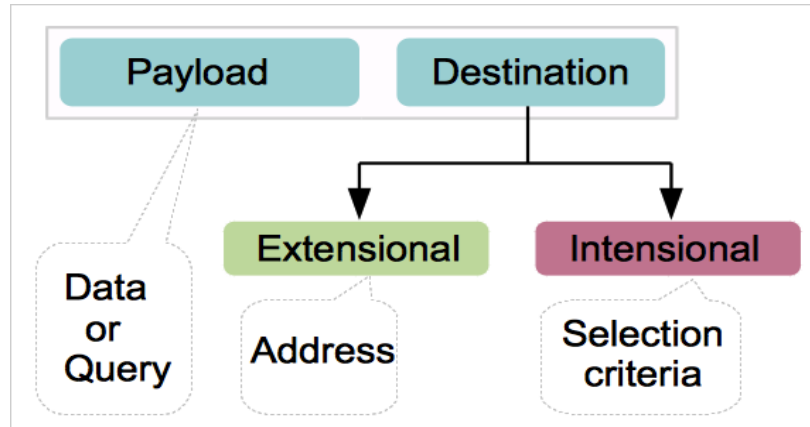


Figure 3.1: Message model

Definition 3.1. A destination is defined as a pair formed by (i) an intensional destination, given by a query, whose result is of type node *Id*, and (ii) an extensional destination, which consists of a node *Id*.

Having defined the destination, we thus can notice that a message may contain either a query in the destination, or a query in the payload, or queries both in the destination as well as in the payload. 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 (e.g. local computation). Only if a node satisfies the *dest-query*, is it authorized to read and compute the *content-query* which might be complex. Interestingly, this distinction allows to optimize the distributed computation of queries.

For an application where the destination nodes are intensionally 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.

3.2 Destination Execution Priority Order

Having defined extensionally and intensionally the destination of messages, two ways of execution are possible. When receiving a message, either the extensional destination or the intensional destination is first evaluated.

We distinguish between two modules performing the algorithms related to the evaluation of the destinations of messages on nodes: (i) Reception module that receives messages from the network, and (ii) Emission module that sends messages to other nodes in the network. We next define the priority order of execution with examples.

3.2.1 Extensional destination higher priority order

We next treat the case where the extensional destination has the highest execution priority order. Two cases have to be considered: (i) when a node, say α , receives a message, and (ii) when node α

3.2 Destination Execution Priority Order

has messages to send.

Let us treat the first case when a message is received. Node α first checks the (extensional) destination. If the extensional destination is equal to the node address, then the node treats the payload. Otherwise, node α is not the destination, and the message is transferred to the emission module.

Let us treat the second case when node α has messages to send. For each message, node α using a routing protocol fetches the next hop to the destination from the routing table. If the next hop is reachable, then the message is sent to the next hop towards the destination. Otherwise, instead of discarding the message due to destination failure, node α executes on the fly the intensional destination, gets as a result a new extensional destination, and then routes the message to the new extensional destination. This requires of course that the routing table is maintained by some routing protocol, but it is independent of the choice of the protocol.

Let us take for instance an example of peer-to-peer application such as Chord [148] or VRR [34]. In such systems, nodes are organized in a virtual ring in order of increasing identifier. The topology can change dynamically. Nodes may join or leave the network. The virtual ring is maintained either periodically or upon detection of failure. According to the application, each node is responsible for an interval of keys. A lookup function is used to map a given key to a unique node in the network. A key is the hashing result of a data such as an *Id*, an *address*, or a *document*. Data are encapsulated in messages and sent through multi-hop routing to responsible nodes to which the keys' maps.

In Chord [148], responsible nodes are the nodes that are equal or follow the keys in the virtual ring. Suppose node α , as shown in Figure 3.2(a), has data to send. After hashing the data, node α finds that the responsible node is node γ . Then, a message *msg* with extensional destination γ will be sent from source node α to responsible node γ . Suppose that node γ leaves the network before receiving the message, thus the message *msg* will be lost on intermediate node, say β , due to the detection of γ 's failure (e.g. node γ leaves the network and therefore no route to the destination γ exists).

However, by specifying intensionally the destination, the message *msg* has as extensional destination γ and an intensional destination "*select node Id that is equal or follow the given key*". Upon the detection of γ 's failure, node β evaluates on the fly the intensional destination on local data, gets a new extensional destination, and afterwards sends the message *msg* to new extensional destination. As we notice in Figure 3.2(b), the new responsible node is δ , then the message *msg* is forwarded to δ . This ensures the persistence of data in messages, as well as the resilience of the system supporting the applications.

3.2.2 Intensional destination higher priority order

We next treat the case where intensional destination has the highest execution priority order. As in Section 3.2.1, two cases have to be considered: (i) when node α receives a message, and (ii) when node α has messages to send.

Let us treat the case when a message is received. Node α evaluates the intensional destination. The computation result is a set of node addresses. If α exists in the set, then it evaluates the payload of the message. Otherwise, node α discards the message.

Let us now treat the case when node α has messages to send. For each message *msg*, the intensional destination is evaluated. The computation result is a set of node addresses. The extensional destination of the initial message *msg* is updated by the node addresses in the set. Intuitively, the number of messages obtained is equivalent to the cardinality of the set of results. Afterwards, the

3.2 Destination Execution Priority Order

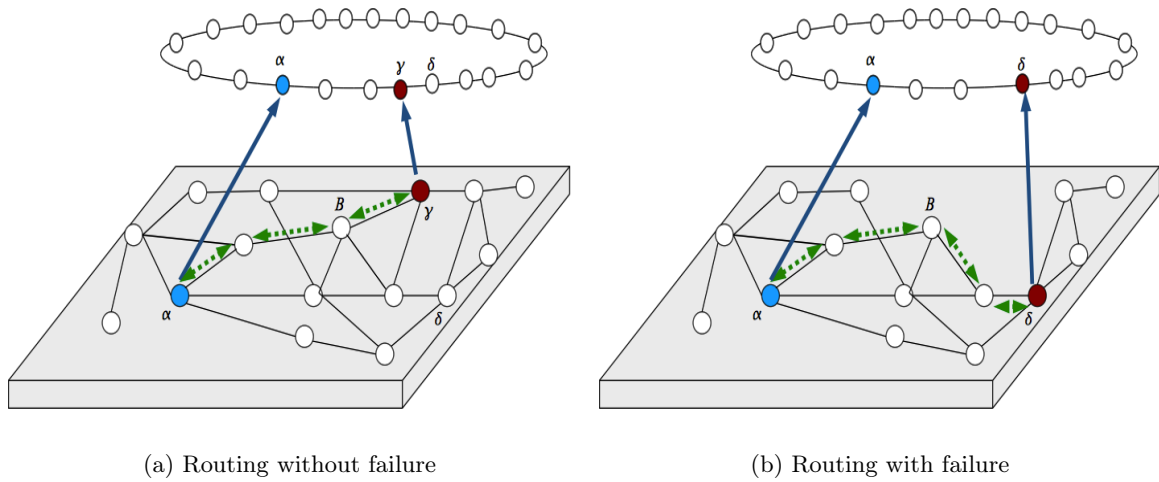


Figure 3.2: Virtual ring on physical network topology

messages are sent to their extensional destinations.

Let us take an example of application in wireless sensor networks. In such networks, thousands of small sensor nodes can be quickly deployed in a vast field to monitor some parameter in an environment. Sensors collect data and then relay streams of data to a common static sink node. To reduce the delay, to balance the load, and to minimize the traffic cost, solutions based on clustering [105, 106, 158, 146, 70] have been proposed. Nodes in a network are grouped into clusters as seen in Figure 3.3. Each cluster head is elected either randomly or by sensor nodes following some properties (e.g. maximum degree, maximum energy). Members send their collected data to the cluster head, which aggregates the data and then sends the result to some sink node.

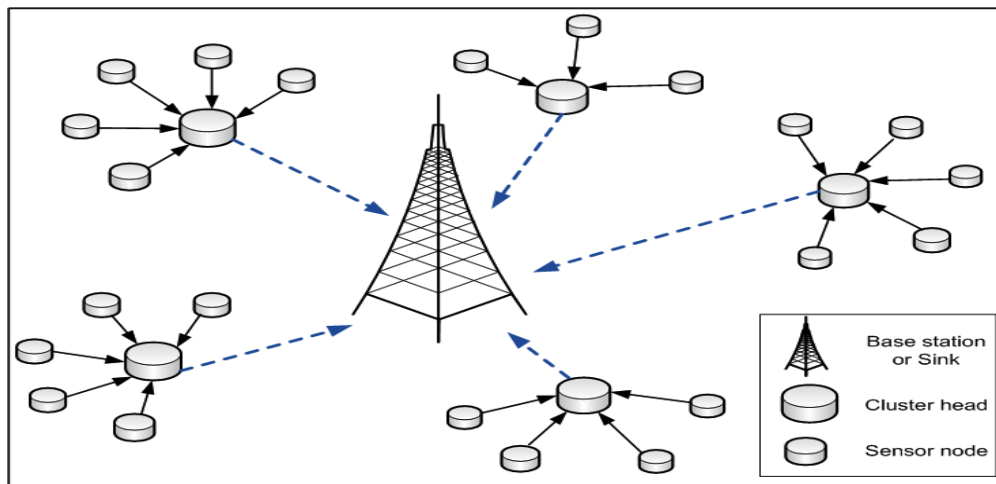


Figure 3.3: Cluster-based wireless sensor network

Usually, sensor nodes exchange messages in order to choose cluster heads which also need to notify periodically all members. For optimization reasons, suppose that the cluster heads do not send

3.3 Intensional Destination Strategies

notifications to members and the cluster heads are not known *a priori* to all sensor nodes. Suppose that the cluster head is specified intensionally on each sensor node using a *selection criteria*. In particular, the cluster head is the sensor node that has the maximum degree, where the degree is the number of neighbors known at each sensor node.

When a sensor node has collected data to send, it encapsulates the data in a message *msg* with intensional destination "*select the Id of the node that has the maximum degree*". To send the message, it evaluates on local data the intensional destination selection criteria, obtains as a result an identifier of a sensor node say β , and afterwards sends the message to β .

When a node receives a message, it first evaluates the intensional destination and checks if it is the cluster head. In particular, when node β receives the message, it evaluates the intensional destination, gets as a result "*x*", and afterwards checks if the result "*x*" is equal to its address β . If it is the case, it processes the message. Otherwise, it sends the message to the cluster head "*x*".

3.3 Intensional Destination Strategies

The destination of messages is specified both extensionally and intensionally. When a message is traveling in a network, it may concern all nodes or a subset of nodes. We distinguish in this section between two strategies: (i) a node may treat the message and at the same time send the message to neighbors, or (ii) a node may treat the message and afterwards decide how to manage the message. We next present the two strategies.

3.3.1 Message decision before processing

We first consider the case where a node, say α , manages received messages before processing them. When node α receives a message, it treats the message and at the same time sends it to other nodes (specifically to neighbors) to be evaluated. Using this strategy, all the nodes in the network receive and treat messages sent.

Let us take an example of applications of social networks where a user, u , sends advertisement messages (e.g. profile) to sets of users that are unknown *a priori*. The user u sends messages with destination specified intensionally addressed to all interested users. Then every user in the network treats the messages.

Note that in our model, a message might contain both a *content-query* and a *dest-query* as we have seen in Section 3.1. In this case, the destination of messages is specified intensionally, by *dest-query*, and all nodes in a network evaluates then the messages. However, an important characteristic of this kind of messages is that not all nodes evaluate the *content-query*. Only nodes that satisfy the *dest-query* are authorized to evaluate the *content-query*. The latter might be complex and require a distributed program to be evaluated. Interestingly, this strategy together with messages including both *content-queries* and *dest-queries* allow the distributed computation of *content-queries* on only interested nodes.

3.3.2 Message decision after processing

We next consider the case where a node, say α , first treats received messages and then decides how to manage them. When node α receives a message, it evaluates the intensional destination. For simplicity, we consider that the computation's result is always a set of node addresses. Based on the set of results, node α then decides how to manage the initial message following different strategies.

3.4 Intensional Destination Specification

The decision can be made on each node based on local data, taking into consideration the node's properties such as energy, memory, complexity, etc. We next describe briefly some of the strategies that can be used.

- **Treat the message:** Node α evaluates the payload of the message.
- **Discard the message:** Node α discards the message. For instance, node α has a low level energy, and thus prevents processing messages in order to maximize node's lifetime and consequently the longevity of the network.
- **Create messages and send them in multicast or unicast mode:** Node α has information concerning the possible destinations of the received message. The destinations might be the powerful nodes in the network, or the nodes that satisfy certain properties. Then node α creates new messages with extensional destination the identifier of the nodes in the set of results, and then sends the messages in unicast or multicast mode.
- **Send the message to neighbors:** Node α has no information concerning the possible destinations of the received message. Thus, it sends the message to neighbors.

3.4 Intensional Destination Specification

The intensional destination is specified by a selection criteria which is application-dependent. The selection criteria is defined upon application specification. It may be for instance a very simple property such as the type of a node (e.g. sink, sensor, etc.) or based on local data (e.g. local SQL-like query, etc.), or it can be more complex following a distributed program. In this section, we propose two specifications: (i) a SQL query executed locally on a node, or (ii) a query specified in a high-level data-centric programming language, Questlog [12], which allows to express complex strategies to pull distributed data.

3.4.1 Intensional destination as SQL query

The intensional destination selection criteria is specified by a SQL query. It is executed on local data of nodes. A local data structure that belongs to intensional destination query should be defined upon application specification. For instance, suppose a wireless sensor network application where some sink node fires a query to collect the identifiers of sensor nodes that have a temperature greater than a threshold T . In this case, the selection criteria is based on the temperature. A local data structure *IntDestination* with two attributes *NodeId* and *Temperature*, as shown in Table 3.1, is defined and installed on each sensor node.

Table <i>IntDestination</i>	
NodeId	Temperature
varchar	int

Table 3.1: Data structure for intensional destination

Having defined the corresponding intensional destination data structure, the intensional destination SQL query is specified as shown in Listing 3.1.

```
SELECT NodeId FROM IntDestination
WHERE Temperature > T;
```

Listing 3.1: Intensional destination as a SQL query

3.4 Intensional Destination Specification

The SQL query is executed on local data, which is updated either by the node (device) itself or from other nodes. Dedicated programs to maintain data related to the intensional destination may increase the communication and computation complexity. Nevertheless, they could be maintained using application or routing data without increased overhead.

3.4.2 Intensional destination as Questlog query

The intensional destination is specified by a Questlog query expressed in a high-level data-centric language *Questlog*, which will be presented in details in Chapter 5. *Questlog* is a rule-based language of the form:

$$head : -body$$

well-adapted to complex applications queries as well as to reactive protocols. *Queries* in Questlog are based on the relational model, and are of the following form:

$$?R(@x_1, \dots, x_\ell)$$

where R is a *relation symbol* of arity ℓ , and $x_1, \dots, x_\ell$ are variables or constants. The attribute prepended by the symbol $@$ represents the destination to where the query should be sent. Queries are associated to rule programs which define their semantics. A rule program is composed of a set of rules that are evaluated in parallel. The program is installed on each node of a network.

When a node receives a message, it evaluates the intensional destination expressed in a Questlog query. The corresponding rules are retrieved from local data store of a node. The node applies each of them. For each rule, two cases have to be considered according to the body which may contain a subquery.

- the body is *simple* with no subquery included, it is then evaluated locally on the node;
- the body is *complex* with subquery included, the subquery is then sent to the appropriate node.

Some bookkeeping is performed to keep track of pending queries and the corresponding subqueries. When answers are received, the pending query can be resumed, and then obtain results.

Let us take the same example of wireless sensor network application, presented in Section 3.4.1, where some sink node collects the identifiers of sensor nodes in a network that have a temperature greater than a threshold T . The Questlog query for such application is defined as follow:

$$?WarnId(@x, t)$$

where x is the identifier of a node, and t is a variable that corresponds to the temperature. This query is fired by the sink node and sent in a message to all nodes. This query is associated with a program that defines its semantic. The following program is used to evaluate the query.

$$\uparrow WarnId(x, t) : - Tmp(x, t), t > T. \quad (3.1)$$

We suppose that each node has the relation Tmp and it is used to save the temperature of the corresponding node. When a node, say α , receives a message with intensional destination query $?WarnId(@x, t)$, it uses Rule (3.1) to evaluate the query. If the temperature t of node α is greater than T , then the identifier of node α as well as its temperature will be sent to the sink. This is the meaning of the symbol " $\uparrow$ " in the head of a rule.

3.4 Intensional Destination Specification

Questlog allows to reformulate queries, and express complex strategies to pull distributed data. It is well-suited for distributed applications running over networks with no knowledge on the location of data.

The use of Questlog to express the intensional destinations is (i) to provide high-level abstraction for the destination of messages, (ii) to reformulate complex queries, (iii) to program applications that adapt dynamically to their environment in a reactive manner, and (iv) to provide modularity to ease altering and modifying the code. In Chapter 5, we present in details the Questlog language as well as its operational semantics.

Conclusion

In this Chapter, we proposed a framework that offers a high-level abstraction for the destination of messages. This framework provides a new model of messages whose destination is specified both extensionally, by an address, and intensionally, by a selection criteria. We then specified two ways of execution for the destination of messages. When a message is received, either the extensional destination or the intensional destination is first evaluated. We defined two strategies of manipulating messages with intensional destinations. A node may (i) treat the message and at the same time send it to other nodes, or (ii) treat the message, and then decide how to manage the message based on the set of computation's results. Finally, we proposed to specify the intensional destination either by a SQL query executed locally on a node, or by a query specified in data-centric language, Questlog, that allows to handle intensional destinations and program complex strategies to evaluate them.

In the next chapter, we demonstrate the framework in a special class of peer-to-peer systems, where the communications between peers is based on extensional/intensional destination, using extensional destination higher priority order. While in Chapter 7, we demonstrate the framework in the domain of wireless sensor networks, where we develop a dynamic clustering protocol based on extensional/intensional destination, using intensional destination higher priority order.

Seamless Distribution of Client/Server Applications

4

Contents

Introduction	39
4.1 Client/Server Application	41
4.1.1 Considered Applications	41
4.1.2 Restrictions on Applications	42
4.1.3 Online Multi-player Game Application Example	42
4.2 Distribution Model	45
4.2.1 Distributed Hash Tables	45
4.2.2 Data Distribution	46
4.2.3 Query Distribution	48
4.3 The Netlog language for distributed protocols	49
4.4 Data centric overlays	52
4.4.1 Distributed lookup	52
4.4.2 Data replication	57
4.4.3 Routing	60
4.5 A Distributed Server for a multiplayer game	62
Conclusion	64

Introduction

Peer-to-peer (P2P) systems have been widely used to alleviate the burden of servers, by transferring to peers in a network tasks that do not require a centralization of the information. Their architecture can be more or less structured, with nodes¹ playing identical or different functions, and with or without interaction with a centralized server. They have been very successful in various fields such as file sharing (e.g. Napster, Gnutella), and communication networks (e.g. skype). A wide range of applications are now emerging over P2P systems, such as social networking [33, 104], multiplayer games [82, 68], mobile messaging [129], video broadcasting [92], etc.

Most of these applications are essentially data centric, they rely on exchange of data pushed and pulled by the peers, which can be modeled as queries over a database. In this chapter, we develop an environment which supports jointly data centric applications simply coded in a client/server architecture, together with routing and indexing protocols coded in a declarative setting. This environment distributes the applications' data and queries, and uses as shown in Chapter 3 messages with *extensional/intensional* destination to ensure persistence of *payload* in messages under changes in the network.

In a client/server setting, the clients have views over a centralized database, they can update their views (client actions), while the server can perform updates over the whole database (system actions), triggered by timers as well as by clients' actions. We demonstrate that under some restrictions, such applications programmed as a collection of queries over a database, can be distributed seamlessly, that is without changing the initial queries, from a client/server architecture, to a P2P architecture with the appropriate overlay.

Numerous techniques have been developed to support peer-to-peer overlays, such as Chord [148], or Pastry [143] for instance. As distributed algorithms in general, they require high programming skills, and their correction is very difficult to guarantee. Following the trend opened by declarative networking [96, 94], we define our overlays using declarative data centric programs with the Netlog language [66], thus resulting in a fully data centric modeling of the application.

In a client/server system, the server handles the application data and queries. In our setting, the server is distributed on nodes of the P2P system the database fragmented, and client queries should be sent to the nodes responsible for the corresponding fragments of the initial database. The data is fragmented horizontally over the peers. For efficient distribution of data and queries, we rely only on the *unicast* mode. Some assumptions are thus necessary to be able to identify the responsible nodes. We assume that each table has (at least) an index attribute. The values of index attributes are mapped to hash keys using a hash function. The corresponding horizontal fragments of the relations are stored on their responsible peers. Some restrictions are imposed on the queries to ensure smooth distribution through the P2P system, making full use of the distributed hash table (DHT). Queries should in particular have at least one *where* argument on an index attribute, which should be instantiated. Moreover join should be performed on index attributes as well. Under such restrictions, we show that the distribution can be done smoothly using the DHT protocols.

Our system relies on the Netquest virtual machine, initially proposed in [66] to evaluate Netlog programs. The Netquest virtual machine is coupled with an embedded Data Management System, DMS, which stores all the data as well as the bytecode of the Netlog programs. The bytecode is obtained by a compilation from Netlog into an SQL dialect. The Virtual machine makes calls to the DMS to evaluate the bytecode of the Netlog programs, which result in updates of the database, and production of messages. Taking full advantage of messages model shown in Chapter 3, we extend

¹In this chapter, we use peer and node interchangeably

the Netquest machine to handle messages with intensional addresses to peers that perform server duties, and ensure persistence of data traveling even in case of changes in the network due to a node failure or departure.

The Netquest machine has been shown to be portable over small devices, as long as they support an embedded DMS [25]. It runs as well on the QuestMonitor platform [29], which allows to monitor the communication between peers, the evolution of the local data stores, as well as the execution of the declarative code. Moreover, proof techniques have been developed in Coq to certify Netlog programs [56].

We illustrate our technique over online multiplayer games, which are starting to emerge over P2P environments [82, 68]. This type of application relies on a scenery from a virtual world, which constitute static data with graphical properties that are out of the scope of this work. Games also involve mutable objects, whose properties can be updated, and avatars representing the players, that can change their attributes. As for other applications of interest to this work, e.g., social networks, sharing, email services, etc. the clients can access data of interest to them through views. The clients can perform actions, which consist in updating these views (e.g. moving an avatar means updating its position), while the system can perform more general actions such as updates over the whole data. Most of the actions of such games can be captured in a purely data centric perspective, even if like for other applications, additional characteristics are important, such as trust and security issues [40], as well as real time aspects, essential for communication systems [44].

As an example of interest, we consider the online multiplayer game "*Auction market*", which can be defined by sequences of queries over a central database. Each player has access to views giving the data pertaining either to its avatar in the game or to global data. A player participates to an auction by making a bid, while the system at the expiration of the auction, changes the owner of the object, and updates the bank accounts of both the seller and the buyer according to the price.

So far as the network is concerned, we make the following assumptions on the application. We assume that players participate to the game over a network to which they connect through devices in some short range communication mean (e.g. bluetooth). The players thus form an ad hoc mobile network. They can physically enter or leave the network, as well as move from one place to another, without being disconnected from the application. The players form a pure P2P system, with nodes playing identical roles, and no centralized server. The overlay network is formed by a distributed hash table, DHT, which is used to distribute both data and computation initially performed by the server.

For the game application, we propose two implementations of DHTs (Chord and Ring) constructed over a routing table defined and maintained by a DSDV like protocol [126]. DSDV is a table-driven routing protocol based on the Bellman-Ford algorithm, well adapted to ad hoc networks. For reliability purposes, we propose as well two implementations of protocols to replicate data on other nodes. The protocols can be chosen with respect to the applications. They can be simply modified or even changed without altering the corresponding applications. They are designed, together with the extensional/intensional destination, to guarantee that the game can run smoothly over a network where nodes can fail. Formally, we concentrated on protocols that support the failure of one node at a time. More dynamics would impose to revisit the replication strategy.

Our experiments with the game are presented in Section 4.5. The scenario, with a node joining the network, participating to the game, moving physically while playing, and leaving the network before the distributed server handles the updates in the game, shows the robustness of the proposed protocols in Netlog. Our experiments show that the movement of one node does not affect the game, all data are preserved, and the duties ensured by the leaving nodes are distributed to other nodes.

4.1 Client/Server Application

The Chapter is organized as follows. In the next section, we explain the considered client/server applications as well as the restrictions on queries, and we present an example of an application of interest. In Section 4.2, we present our distribution model for data and queries over an overlay. In Section 4.3, we present the Netlog language together with the virtual machine to evaluate Netlog programs. Section 4.4 is devoted to fundamental protocols supporting the overlay. In Section 4.5, we illustrate over an example the protocols to distribute the applications specified as centralized queries, and monitor the behavior of these protocols over the QuestMonitor platform.

4.1 Client/Server Application

In this section, we first present a model of applications that can be programmed as updates over views in a client/server setting. We then describe the restrictions imposed on applications queries and schemas to allow efficient distribution, and finally show an online multiplayer game as an example of an application of interest.

4.1.1 Considered Applications

We consider applications which can be described as sequences of database updates performed by clients over a centralized server. The server stores all the data from all clients, while the clients can access and modify only some part of these data.

Applications queries are specified using Structured Query Language (SQL). In all these applications, the clients can access data of interest to them, which can generally be defined by views over some horizontal fragments of the data structures. The clients can perform actions, which consist in querying or updating these views (e.g. moving an avatar means updating its position), while the system can perform more general actions such as queries and updates over the whole data. Such applications can easily be modeled over database management systems, in a client/server framework, with:

- *Client views*: over the global database, allowing access to their account as well as to public data;
- *Client actions*: over their views, modeling their changes over their data and public data, and interactions with other clients;
- *System actions*: over the global database, triggered and executed by the server.

The *views* offered to the client include their account and data as well as public data. They are defined as conjunctive queries over the global schema. We denote views by names prefixed with a "V", such as *VTableName* for the table *TableName*. We suppose that views have the same structure as their corresponding tables in the global schema (except user attribute).

The *client actions* include updates (insert/delete/update) of client data. They correspond for instance to moving an avatar, shooting at others in multiplayer games or inserting, deleting contacts in social networking, etc. The client actions are modeled as update conjunctive queries over their views. To improve the realtime feeling, the client actions can be restricted to updates on the views, which will be reported as an update on the server over the global schema. A trivial reformulation of the query, such as replacing view table name (*VTableName*) by initial table (*TableName*), will be done locally on each node before sending the query to accomplish the updates on the server. Some of the attributes might be system defined (e.g. account, balance, current time, etc.). An avatar, for example, cannot retrieve the account balance of another avatar neither change the balance of its bank account.

4.1 Client/Server Application

The *system actions* are update queries on the server, generally triggered by timers or client actions. They are modeled as update conjunctive queries over the global schema. System actions are performed by the server. They are represented by a set of queries stored on the server.

4.1.2 Restrictions on Applications

In a client/server setting, application queries are sent to the server that holds the data. In a distributed setting, the applications queries need to be sent to the peers responsible for fragments of the initial database. Our setting distributes seamlessly client/server applications into P2P systems. To allow efficient distribution of the server and application queries, we rely only on the unicast mode. Some restrictions are thus imposed on application queries as well as on application schemas. This is needed to specify the placement where application data and queries will be stored and executed on distributed systems, respectively.

We propose to use *index placement*. Each table should have at least one *index attribute* specified *a priori* upon writing the client/server application. Note that the *index attribute* might be the *key attribute* or any other attribute of the related table.

In our specification, a class of *allowed queries* is introduced. Allowed queries involve *select/insert/delete/update* queries, which might contain joins. This prototype permit neither *nested* queries, nor *distinct*, *group by*, *order by*, and *union constructs*.

We restrict the class of allowed queries for efficient distribution of the initial client/server queries to the relevant peers of the distributed system. Queries should have at least one *where* argument on an index attribute. Intuitively, index attributes need to be (fully) instantiated in the (update) queries, and join conditions need to be on index attributes, to allow a distribution relying only on *unicast* of the queries to a well defined set of peers.

We next discuss the restrictions of the different types of allowed queries. Let us start with the *select* query. We distinguish between two kinds of *select* queries:

- *select mono-table*: a select query that accesses at most one table;
- *select multi-table*: a select query that accesses at least two tables.

We impose on queries of type *select (mono-table or multi-table)* to have at least one index attribute instantiated, while update queries of type *insert/delete/update* should have all index attributes instantiated.

Queries might contain join. In this case, the join should be defined on index attributes, where left and right operands are used as index attributes. We permit queries which have partially instantiated index attributes. It will be handled by recursively decomposing the related query into allowed simple subqueries. Then, the set of index attributes is transitively defined. More precisely, an answer of a subquery instantiates other index attribute(s) of the initial query.

We formally define the restriction on queries as follows:

Definition 4.1. A *select (resp. insert/delete/update)* query is allowed if at least one of (resp. all) its index attributes are instantiated by values, and the joins are defined on index attributes.

4.1.3 Online Multi-player Game Application Example

In this section, we consider an example of multi-player game that can be described as sequences of database updates performed by clients over a centralized server. The server stores all the data from all clients, while the clients can access and modify only some part of these data, which can be defined by views over the whole data.

4.1 Client/Server Application

Online multi-player games over virtual worlds, such as Second Life, World of Warcraft, etc. constitute fundamental applications of this type. Currently, most massively multiplayer games are implemented on a client/server architecture, with a server which handles both client accounts and game states. Various types of clusters are used, for scalability purposes, to support massive numbers, millions, of players at the same time.

If we leave apart the graphical interface in which players evolve, the basic actions they perform can be modeled easily as database updates. Clients generally participate to the game through an *avatar*. The game relies on some stable "landscape", which can be seen by the clients, using their views over the global data. Most games support mutable virtual objects, which can be changed (created, destroyed, exchanged, etc.) by the players during the course of the game.

The server knows at every moment the connected clients, as well as the updates they make on the data (e.g. creating, deleting or moving avatars, exchanging virtual objects, etc.). We illustrate over an example how each of the basic actions of the payers can be described with a set of queries over the centralized database of the server.

The list of avatars of players is stored in the table **Owner**, which contains an authentication key for each avatar. We consider a simple game in which avatars exchange, sell or buy objects through an auction market. Each avatar owns bank accounts (table **Bank**), can register itself into a market (table **Market**), buy or sell their objects (table **Objects**), into an auction market (table **Auction**). Samples of the tables used in the example are shown below.

Table Owner			Table Bank			
Avatar	Auth	Client	Name	Avatar	Account	Balance
Bob	37	0012	WorldBank	Bob	123985642	1524
Alice	54	0193	GlobalBank	Alice	AB87532	845

Table Object					Table Market		
Class	Avatar	Name	Price	Id	Name	Avatar	Account
Pet	Bob	2 yo Mouse	75	1	Catown	Bob	123985642
					Catown	Alice	AB87532

Table Auction						
Name	Seller	Buyer	OID	Price	ExpTime	MinPrice
Catown	Bob	Alice	1	150	1305273029	100

Each client has a local view of these data, which concerns their avatar. We use the term *self* to denote value characterizing the client. The only actions clients can perform is to query or update their local views. When an avatar updates data in a view, the update is sent to the server to be performed. Systems actions consist of arbitrary queries over the global schema and are triggered and performed by the server. We show below the main actions of a particular transaction of the auction game, which are defined by simple queries. We use variable *self* to denote the client Id.

4.1 Client/Server Application

1. See the auctions on the market (view)

```
INSERT INTO VAuction
SELECT Auction.Name, Auction.Seller, Auction.Buyer, Auction.Old,
        Auction.Price, Auction.ExpTime, Auction.MinPrice
FROM Auction, Auction AS Auction2, Owner
WHERE Auction.Name = Auction2.Name AND Auction2.Seller =
        Owner.Avatar AND Owner.Client = self;
```

2. Propose a new auction (client action)

```
INSERT INTO VAuction
SELECT 'Catown', Owner.Avatar, Owner.Avatar,
        4,100,1305273029,100
FROM Owner
WHERE Owner.Client = self;
```

3. Make a bid (client action)

```
UPDATE VAuction
SET VAuction.Price = '150', VAuction.Buyer = (SELECT Owner.Avatar FROM
        Owner WHERE Owner.Client = self;)
WHERE VAuction.Name = 'Catown' AND VAuction.Seller = 'Bob'
        AND VAuction.Old = 1;
```

4. Cancel a bid (client action)

```
DELETE FROM VAuction
WHERE VAuction.Name = 'Catown' AND VAuction.Old = 1 AND VAuction.Seller
        = (SELECT Owner.Avatar FROM Owner WHERE Owner.Client = self;);
```

5. Conclude an auction when the auction has expired (system action)

```
UPDATE Bank
SET Bank.Balance = Bank.Balance - 150
WHERE Bank.Account = 'AB87532';

UPDATE Bank
SET Bank.Balance = Bank.Balance + 150
WHERE Bank.Account = 'AB123985642';

DELETE FROM Auction
WHERE Auction.Name = 'Catown' AND Auction.Seller = 'Bob'
        AND Auction.Old = 1;

UPDATE Object
SET Object.Avatar = 'Alice', Object.Price = 150
WHERE Object.Id = 1;
```

In their seminal paper, Knutsson et al. [82] showed how such a multiplayer game could be developed over a P2P architecture using the Pastry [143] overlay network. We show how more generally any application described by queries and views as presented above can be distributed seamlessly over an overlay.

4.2 Distribution Model

We consider a network constituted by peers that communicate in a P2P fashion (over the Internet, device to device, etc.). They can join and leave the network at any time and participate to the applications with the other connected peers. Collectively they support the tasks of the centralized server, by storing fragments of the data on peers, and by routing the queries (views, client actions, system actions) to the peers in charge. There is no centralized server, and the peers play the same role.

The application is defined over some *schema*, which will be used in the distributed environment exactly like in the centralized one. As we have seen in Section 4.1.2, each relation has at least one special attribute called *index attribute* (surrounding attributes in the tables of our example), such as *Avatar* in table *Owner*, and *Id* in table *Object*. Attributes of the tables are either client defined (e.g. *Name*, *Price* in table *Object*), or system defined (e.g. *Balance* in *Bank*).

4.2.1 Distributed Hash Tables

The client/server queries modeling the applications will be distributed to P2P systems. Tables in such systems are fragmented horizontally, the fragments are distributed over the peers. There are classical techniques to distribute data such as DHTs for instance which assign peer Ids to data items. Different DHT systems such as CAN [138], Chord [148], Pastry [143], Tapestry [167], OpenDHT [142], etc., can be used to provide an efficient lookup service. As a particular choice, we implemented a mechanism for key lookup, Ring, which maintains a ring of peers. Another mechanism, Chord, has been as well implemented. We describe the implementation of the two mechanisms later in Section 4.4.1. For the Ring protocol, a hash function is used to hash *index attribute values*. The result of hashing index values are keys (hash values). We assume that the hash function ranges over the domain of peer Ids, and that for simplicity the peer Ids are uniformly distributed. A circular order is defined over this domain, used to form the DHT. In our system, a lookup function $\mathcal{H}$ is used to map a given key to a unique peer in the network. Each peer, say α , is responsible for storing the fragment of each relation corresponding to the tuples whose hash value is such that α is the largest node Id smaller than the hash value of the tuple (predecessor).

Peers sometimes may fail. When faults occur in hardware or software, programs (e.g. DHT, routing, etc.) installed on peers may fail before they have completed the intended computation. Several mechanisms such as data replication, can be used to handle failure and to ensure the reliability of data under peers movement, so that peers can leave the network without perturbing the application.

A node say α is responsible for the intervals of keys $[\alpha, s]$ where s is the immediate node Id (successor) that follows the node α . When the node α leaves the network, all the data that have keys between $[\alpha, s]$ will be lost. For reliability, accessibility and quality of service purposes, data on each node can be replicated on different nodes.

There are two simple methods used to replicate data (i) passive and (ii) active replication:

- **Passive replication:** This method involves processing each single request on the responsible node, which afterwards transfers its resultant state to other replicas. For instance, the replication method shown in Figure 4.1(a) consists in sending the data to the responsible node (NR), which in turn saves a copy on its immediate successor.
- **Active replication:** This method consists on performing the same request at the responsible node as well as every replication node. For instance, the replication method shown Figure

4.2 Distribution Model

4.1(b) consists on sending two requests at the same time to the responsible node (NR) as well as to its immediate successor.

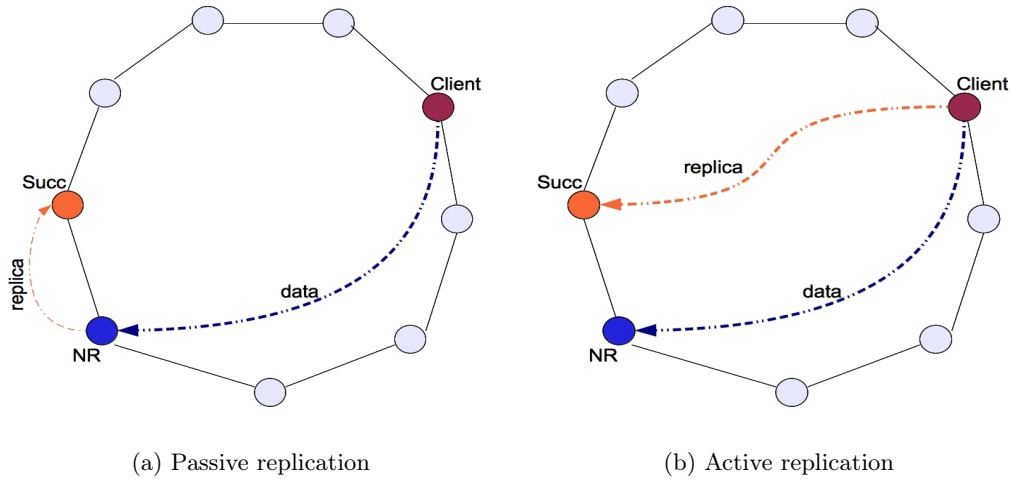


Figure 4.1: Data replication techniques

Intuitively, the active replication is more reliable but also more costly in terms of communication complexity. In our proposed DHT, we realize a single replica in the immediate successor of the responsible node, using the passive replication. For simplicity, we suppose that nodes do not leave the network while in processing rounds. Each node therefore handles received data, and produces and sends derived data in the current round before stopping activity.

Our discussion is based on a single replica in the immediate successor, but can be extended to multiple replicas (e.g. in the n successors) to cope with higher failure frequencies, at the cost of increased bandwidth usage. There is a tradeoff between *reliability* (data always exist under any circumstances and that related to the number of replicas), and *complexity* (number of messages to be exchanged).

4.2.2 Data Distribution

We next present the distribution model of data. Each relation has at least one *index attribute*. For each tuple, the hash result of its index attribute(s) value(s) is mapped by the hash function $\mathcal{H}$ to a peer Id. Let us consider the tuple r :

$$r = [Catown, Bob, Alice, 1, 150, 1305273029, 100]$$

of the table **Auction** shown in Section 4.1.3. Since the table has indexes on attributes *Name* and *Seller*, the tuple is mapped by the hash function $\mathcal{H}$ to the numerical closest predecessor of the following keys: $hash(Catown)$, and $hash(Bob)$.

This process is repeated for all the tuples of the tables used in the related application. In our game example shown in Section 4.1.3, the distribution process involves all tuples of the tables **Owner**, **Bank**, **Object**, **Market**, and **Auction**.

We have seen above how the data can be fragmented and distributed over the peers using a DHT. The main difficulty is to distribute the data (and queries as we will see in Section 4.2.3)

4.2 Distribution Model

to the appropriate peers, and guarantee some persistence in the system. For this purpose, we use our proposed model of messages presented in Chapter 3. Recall that the destination of messages is specified both *extensionally* and *intensionally*. The *extensional* destination is a node Id that corresponds to the responsible node, while the *intensional* destination is a selection criteria defined upon application specification.

In our game example, the intensional destination is specified by a query extracted by a *LOOKUP* function of the system, Listing 4.1. This function is based on a table, *KeysIntvl*, specified upon writing the application.

$$KeysIntvl < NodeId, MinKey, MaxKey >$$

This table stores the responsible keys interval of each node in the DHT. It is maintained by a protocol defined together with the overlay and the routing protocol. In our demonstration with the game, we suppose that the table *KeysIntvl* is maintained by the routing protocol without incremental overhead. Later in Section 4.4.3, we will see that the routing table is updated periodically and each node has routes to all nodes in the corresponding network. Each time the routing table is updated, the table *KeysIntvl* is updated as well. It consists to update the predecessor (*MinKey*) and the successor (*MaxKey*) of each node (*NodeId*) in the table if the latter always exists in the network.

```
SELECT NodeId
FROM KeysInterval
WHERE hash(index attribute value of r) > MinKey
AND hash(index attribute value of r) ≤ MaxKey;
```

Listing 4.1: Intensional destination *LOOKUP* function

Consider again the tuple *r* which has two index attribute values, *Catown* and *Bob*. As we have seen above, this tuple will be distributed by messages to two peers, say *n_i* and *n_j*. For each index attribute value, the system extracts an *intensional destination* query using the *LOOKUP* function as shown in Listing 4.2, and Listing 4.3:

```
SELECT NodeId
FROM KeysInterval
WHERE hash(Catown) > MinKey AND hash(Catown) ≤ MaxKey;
```

Listing 4.2: Intensional destination for index attribute value "*Catown*"

```
SELECT NodeId
FROM KeysInterval
WHERE hash(Bob) > MinKey AND hash(Bob) ≤ MaxKey;
```

Listing 4.3: Intensional destination for index attribute value "*Bob*"

A message consists essentially of a payload, the content of the message which might be data (a tuple) or an application query (as we will see in the Section 4.2.3), together with a destination. Messages are routed according to their extensional destination if a route to that destination can be found. Otherwise, the intensional destination is evaluated, and a potentially new extensional destination is obtained. The failure of the node which is the extensional destination of a message, doesn't result in the loss of the message, whose destination can be recomputed from the intensional destination.

4.2 Distribution Model

4.2.3 Query Distribution

In this section, we show how the client/server application queries are distributed to their appropriate peers, which contain the appropriate fragments of data, to be evaluated. Based on Section 4.1.2, each query has at least one *where* argument on an index attribute, which needs to be instantiated.

We have seen two kinds of *select* queries. We first start by the *select mono-table* query. This type of query does not contain join. Given an allowed query, its index attribute value(s) is retrieved. Based on the hash function, the index attribute value(s) is hashed which results in a key(s). After that the lookup function $\mathcal{H}$ is called to get the responsible peer(s), say α .

At the same time, the intensional destination is extracted as we have seen in Section 4.2.2, based on the index attribute value of the query. Finally, the query is encapsulated in a message and sent to α . In the case of *select mono-table* query with two or more instantiated index attributes, it is enough to distribute the query to be evaluated on only *one* peer. By default, the first index value is selected.

We now show how the *select multi-table* will be sent to the peer in charge of the different fragments. This type of queries contain *join*. The join is handled by decomposing the query, say q , into allowed simple subqueries. Then, they will be distributed exactly like for the *select mono-table* queries. When answers of subqueries are received, the initial query q is rewritten and evaluated locally.

Note that in the current prototype, we didn't implement a *query engine* to manipulate client/server application queries. These queries are transformed into a rule-based program specified in the Netlog language [66] and executed in the Netquest virtual machine, which will be detailed in Section 4.4. We will show in Chapter 8 an overview of the Ubiquest system [7, 8] which includes a particular engine (distributed query engine) that can handle as well such queries.

Let us move on to show how the update queries of type *insert/delete/update* will be distributed to the peer in charge of the different fragments. According to Section 4.1.2, allowed queries have fully or transitively instantiated index attributes. Queries that have fully instantiated index attributes, will be distributed exactly like for the *select mono-table* queries, but to *all* peers Ids obtained by hashing all index attributes values.

Queries that have *join* and have partially instantiated index attributes, will be handled as follows. For each query w , it is handled by decomposing the query w into allowed simple subqueries. The set of index attributes is transitively defined. More precisely, subqueries that have instantiated index attributes will be distributed exactly like for the *select mono-table* queries. Then, answers of subqueries instantiate other index attribute(s) of the initial query w . This will trigger the distribution of subqueries. This process is repeated for each received answer, until distributing the initial query w .

The distribution of queries is done by messages with extensional/intensional destination. The content of messages contain the queries, while the intensional destination is extracted exactly as for data distribution, Section 4.2.2. The distribution of the queries to the appropriate peers is done in two steps:

1. *Extraction of the intensional destination*: The intensional destination query is automatically compiled from the application query, based on the index attribute value;
2. *Routing to the (intensional) destination*: A routing protocol is provided to route messages to their destinations.

4.3 The Netlog language for distributed protocols

Netlog [66] is a rule-based language that allows to express distributed programs. Netlog programs consist of sets of recursive rules of the form *head* :- *body*, where their evaluation follows the *forward chaining* mechanism, in the *push* mode. The rule *head* is derived when the *body* is satisfied. The programs are installed on each node of a network, where they run concurrently. The computation is distributed and the nodes exchange information. The facts deduced from the rules can be either stored on the node on which the rules run, or sent to other nodes. A Netlog rule is triggered by a newly arrived fact, for instance *Hello(NodeId)*, in the *body* of a rule. When a fact is received on a node, all triggered Netlog rules are fired. We next present the language through some fundamental examples of programs for neighbor discovering and network organization.

The following program, Rules (4.1 - 4.4), implements the hello protocol, which is responsible for establishing and maintaining neighbor relationships, using the relation *Link(self, y, t)* where *self* is the Id of the node where the execution is taking place, *s* is a neighborhood address, and *t* is a timeout.

$$\uparrow \text{Hello}(\text{self}) : - \text{TimeEvent}('Hello'). \quad (4.1)$$

$$\downarrow \text{Link}(\text{self}, s, t) : - !\text{Hello}(s), \neg \text{Link}(\text{self}, s, _), \\ t := m_time + m_timeout. \quad (4.2)$$

$$\downarrow \text{Link}(\text{self}, s, t) : - !\text{Hello}(s), !\text{Link}(\text{self}, s, _), \\ t := m_time + m_timeout. \quad (4.3)$$

$$: - \text{TimeEvent}('CheckLink'), \\ !\text{Link}(\text{self}, s, t), t < m_time. \quad (4.4)$$

The **store/push operator** in front of rules, determines where the results are assigned. 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. The **negation** " $\neg$ ", shown in the body of Rule (4.2), is interpreted by local closed world assumption (a fact is not true on a node if it is not stored on that node). The consumption operator, " $!$ ", is used to delete the facts that are used in the body of the rules from the local data store. The *m_time* is the current time machine. The constant *m_timeout* is a metadata defined in the header of the program, and the underscore used in the parameters denotes "any value". A rule, e.g. Rule 4.4, can have no *head* if a consumption occurs in its body.

Periodically using the timer "Hello" in Rule (4.1), each node (say α) sends a *Hello* message to its neighbors. Upon receiving the *Hello* message, neighbors save α in their neighborhood table *Link* with a timeout *t* if α does not exist in the table, in Rule (4.2). Otherwise, they update the timeout of the link to α , in Rule (4.3). Periodically using the timer "CheckLink" in Rule (4.4), each node checks the timeout and deletes expired links to neighbors. All facts prepended by " $!$ " are consumed after the evaluation of the rule.

The following program, Rules (4.5 - 4.7), defines a spanning tree like. The results are distributed so that each node stores the knowledge of its parent in the tree, using the relation *ST(parent, self)*.

$$\updownarrow \text{OnST}(\text{self}) : - \text{Root}(\text{self}). \quad (4.5)$$

$$\downarrow \text{ST}(\diamond y, \text{self}) : - \text{Link}(\text{self}, y, t), !\text{OnST}(y), \neg \text{OnST}(\text{self}). \quad (4.6)$$

$$\updownarrow \text{OnST}(\text{self}) : - \text{Link}(\text{self}, y, t), !\text{OnST}(y), \neg \text{OnST}(\text{self}). \quad (4.7)$$

4.3 The Netlog language for distributed protocols

The **store/push operator**, " $\uparrow$ ", in front of rules, determines where the results are assigned. The effect of " $\uparrow$ ", is to both store the results of the rule on the node where it runs, and push them to its neighbors. The **choice operator** " $\diamond$ " chooses non-deterministically a parent among the possible choices. Note that messages can also be unicasted by adding an "@" in the head in front of a variable denoting the destination.

Assume that $Root(\rho)$ holds on a root node ρ exclusively. With Rule (4.5), the root node ρ derives the fact $OnST(\rho)$, stores it locally and sends it to all neighbors. Each node, say α , that receives facts $OnST(y)$ from their neighbors y , chooses randomly one neighbor as a parent, Rule (4.6). This is done if node α is not on the tree, $\neg OnST(self)$. At the same time, node α stores a fact $OnST(self)$ and sends it to all neighbors, Rule (4.7).

The Netlog programs are evaluated by a virtual machine, Netquest, which runs Netlog programs. 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.

Let us show how a query is compiled into SQL dialect. A query is built for each operator (store, push and deletion) in a rule.

Consider for instance the following rule which contains the three operators. The @ symbol in the body of the rule followed by the variable a denotes the *location specifier*, where the evaluation of the rule is taken place (on node a).

$$\uparrow Link(a, b) : - !Hello(b, @a), \\ \neg Link(a, b).$$

This rule is evaluated when the engine receives a *Hello* message. It is translated into three SQL queries corresponding to each operator.

The first query is the result for the operator push, the second for the operator store and the third for the deletion.

All the keyword beginning by $m_$ (e.g. m_self) are replaced by the engine during the evaluation of the rule. The negation of *Link* is translated with the sub-query into the section *not exists*.

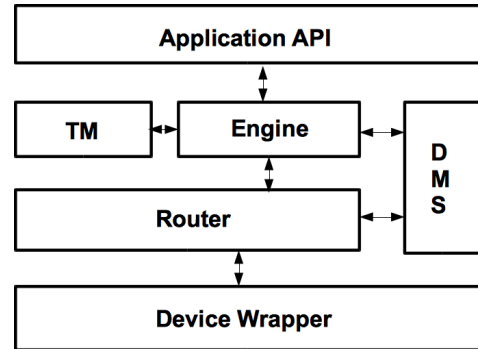
```
SELECT Hello.a, Hello.b
FROM Hello
WHERE Hello.a='m_self'
AND NOT EXISTS (
  SELECT Link.a, Link.b
  FROM Link
  WHERE Link.a=Hello.a AND
        Link.b=Hello.b);

INSERT INTO Link
SELECT Hello.a, Hello.b
FROM Hello
WHERE Hello.a='m_self'
AND NOT EXISTS (
  SELECT Link.a, Link.b
  FROM Link
  WHERE Link.a=Hello.a AND
        Link.b=Hello.b);

UPDATE Hello
SET Hello.deleted=1
WHERE Hello.a='m_self'
AND NOT EXISTS (
  SELECT Link.a, Link.b
  FROM Link
  WHERE Link.a=Hello.a AND
        Link.b=Hello.b);
```

4.3 The Netlog language for distributed protocols

The Netquest Virtual Machine executes the Netlog bytecode 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 realized by Bauderon *et al.* [25] was done in iMote sensors.



The Netquest Virtual Machine is composed of six components:

- Device Wrapper for QuestMonitor: provides an interface to the network. It receives and sends data over the network, and does the address translation between Netlog internal addresses and the network addresses.
- Data Management System (DMS): ensures data storage and query execution. The DMS stores both the data of the applications as well as the declarative protocols running on the system, which are called by the Netlog engine.
- Router: receives and sends Netquest messages through the device wrapper. It chooses in the DMS the best route to reach each destination. If no route to a destination is found, the router evaluates the associated intensional destination query by calling the DMS to find new extensional destinations. We detail below its functionality.
- Engine: evaluates the Netlog programs. If a node receives a message that contains a Netlog fact, the engine loads the rules triggered by the facts of the message and executes them through the DMS, resulting to updates in the database, as well as to messages to send. Its functionality is described briefly in Section 4.4 together with the declarative overlays.
- Timer Manager (TM): manages time events of the system. In particular, Netlog programs can create and manipulate timers. These timers are managed and fired by this module.
- Application API: this module is an interface between the virtual machine and applications. An external application can use the Netlog Virtual Machine to send and receive messages over the network.

Followed our proposed framework, seen in Chapter 3, where we define a new model of messages whose destination is specified both extensionally and intensionally, we extend the router of the Netquest virtual machine to handle the new model. We follow the strategy where the extensional destination has the highest execution priority order.

The Router handles the incoming and outgoing messages. When receiving a message by a device (node), the router checks if the node belongs to the (extensional) destination of the message. If it is the case, the content of the message (payload) is transferred to the Engine. Otherwise, it searches the next hop to the extensional destination and sends the message to it. Otherwise, if the extensional destination is empty or the next hop to extensional destination cannot be found, the router evaluates the intensional destination against the local DMS on the local store of the node, gets as a result a new extensional destination, and routes the message as above. As we will see, the intensional destinations reveal very useful to handle the destination obtained by the DHT, which can have left the network, and can be recomputed on the fly.

The engine loads rules from Netlog programs matching the facts contained in the payload and then evaluate these rules using the DMS. The DMS can update or delete data and create messages

4.4 Data centric overlays

to be sent. These new messages are sent to the network through the device wrapper.

The Engine does not execute directly the bytecode. It orchestrates the tasks to be done to treat messages and facts, following a distributed fixpoint semantics. When receiving facts, a new round starts and a first stage is executed. In this stage, the engine loads and executes rules triggered by these facts. If there are derived facts produced by the engine, a new stage is executed recursively again with these new facts. A round is finished when there are no new derived facts. At the end of a round, produced messages are sent to other nodes in unicast or broadcast mode.

4.4 Data centric overlays

Netlog is well adapted to the development of networking protocols. We next present the overlays which are specified using the *Netlog* language [66]. The overlays are defined by a combination of a DHT protocol, together with a replication as well as with a routing protocol. We show how the basic protocols required for our application to distribute the server tasks over a DHT can be written.

Note that the overlays protocols are independent of the application, and other choice can be easily made. Any protocol offering the functionality of the distributed hash table, like load balancing, managing the hash keys across nodes, as well as routing, could be used. The protocols presented below are chosen for an application running on nodes that form an ad hoc network, which is resilient under node departure.

4.4.1 Distributed lookup

In the following, we present two distributed lookup protocols. We choose to implement the Chord protocol [148] that provides a scalable P2P lookup by reliably mapping a given key to a unique node in the network. We propose also for simplicity an implementation of a distributed lookup protocol *Ring*, that is a circular order defined over the node Ids. Each node then constructs the segment of keys it is responsible for. The network can dynamically change, nodes can join or leave the network at any time.

Chord protocol

We next present the implementation of the Chord distributed hash table [148] expressed in Netlog. For brevity, we present only the main rules. The whole code with 43 rules, is comparable to the program presented in [97], can be found in [149].

Chord is a distributed lookup protocol that given a key, determines the node responsible for that key. Identifiers of nodes are ordered on an identifier circle, called *Chord ring*. Chord maintains the ring and routes efficiently on it. Figure 4.2 shows an example of a Chord ring. Each node in the Chord ring has a unique node identifier, represented by integers in our system. Each Chord node is responsible for storing objects within a range of key-space. This is done by assigning each object using a hash function with key k to the first node whose identifier is equal to or follows k in the identifier space. This node is called the successor of the key k . In Chord, data items and nodes are mapped into the same identifier space. Therefore each node also has a successor, which is the next-higher identifier. For example, the object with key 15 is served by node 16.

In Chord, each node maintains the identifiers (IP addresses) of multiple successors to form a ring of nodes that is resilient to failure. Once a node has joined the Chord ring, it maintains a list of

4.4 Data centric overlays

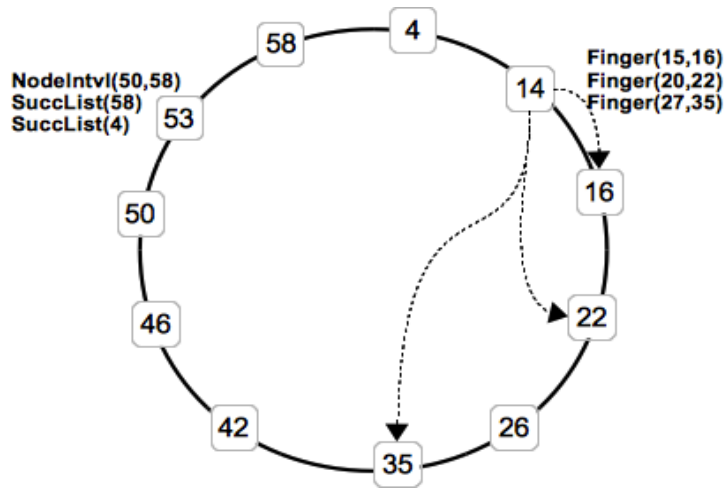


Figure 4.2: A Chord ring

s successors in the ring (the *SuccList* table) with the closest identifier distance to the node, and a single predecessor: the address of the node whose identifier just precedes the node. Each node maintains its immediate successor, called best successor, whose identifier is the closest among all the successors to the current node. For example, if $S = 2$, the successors of node 53 in Figure 4.2 are 58 and 4, its best successor is 58 and its predecessor is 50.

In order to perform scalable lookups, each Chord node also holds a finger table, pointing at peers whose identifier distances exponentially increase by powers of two from itself. The entries in the finger table are used for efficiently routing lookup requests for specific keys. In Figure 4.2, node 14 has finger entries to nodes 16, 22 and 35, as denoted by the dotted lines.

Let us now move on to introduce the corresponding schemas of main tables that have been used to implement the Chord protocol. They are shown in Table 4.1.

Schema	Description
NodeIntvl(p,s)	NodeIntvl(predecessor, successor)
Lookup(x,y,z)	Lookup(destination, sender, key)
LookupRes(x,y,z)	LookupRes(toSender, key, nodeResponsible)
Notify(x,y)	Notify(destination, sender)
Finger(x,y)	Finger(key, nodeResponsible)
SuccList(x)	SuccList(successor)
BestPred(x,y,z)	BestPred(predecessor, sender, key)

Table 4.1: Main schemas of the *Chord* protocol

In addition, there are other tables such as *ChoseNgb*, *AskSucc*, *MyPred*, *FixEntry*, ect., that are used to store intermediate state in our Chord implementation. In the following, we demonstrate how different aspects of Chord can be specified in Netlog: joining the Chord ring, finger maintenance, and ring maintenance.

When nodes start, they setup their initial state. At initialization phase using the timer 'Ini' for only once, Rule (4.8), each node sets its predecessor "p" to 'nil' indicating that there are no

4.4 Data centric overlays

predecessors, its successor "s" to *self* node address, Rule (4.8), and saves them in the relation *NodeIntvl*(*p*,*s*).

$$\downarrow \text{NodeIntvl}('nil', self) : \neg \text{TimeEvent}('Ini'). \quad (4.8)$$

A Node, say α , joins the Chord ring, which was initiated at a node β , by sending a *Lookup* request to node β in the Chord ring, Rule (4.9) and (4.10). If the *Lookup* request succeeds, either Rule (4.11) or (4.12), then node β unicasts a lookup result to α . The latter updates its successor upon the reception of the lookup results, Rule (4.13).

$$\downarrow \text{ChoseNgb}(\min(z)) : \neg \text{Neighbor}(z), \quad \text{NodeIntvl}('nil', self), self \neq 0. \quad (4.9)$$

$$\uparrow \text{Lookup}(@x, self, self) : \neg \text{ChoseNgb}(x). \quad (4.10)$$

$$\updownarrow \text{LookupRes}(@x, k, s) : \neg \text{Lookup}(self, x, k), \quad \text{NodeIntvl}(p, s), x \neq s, k \leq s, k > self. \quad (4.11)$$

$$\updownarrow \text{LookupRes}(@x, k, s) : \neg \text{Lookup}(self, x, k), \quad \text{NodeIntvl}(p, s), k > self, s \leq self. \quad (4.12)$$

$$\downarrow \text{NodeIntvl}(p, s) : \neg \text{LookupRes}(self, k, s), \quad \neg \text{NodeIntvl}(p, s1), k == self. \quad (4.13)$$

A Chord node holds a *finger table*, pointing to nodes whose Id distances exponentially increase from itself, Rule (4.16). An entry of the finger table is specified using Rule (4.14) and (4.15). The constant value *S* is a metadata that defines the size (number of entries) of the finger table. Periodically, each entry in the finger table is updated by firing a lookup request for the related key. Upon receiving a lookup result, in Rule (4.17) and (4.18), the responsible node of the key is updated.

$$\downarrow \text{FixEntry}(1) : \neg \text{FixEntry}(i1), i1 == S. \quad (4.14)$$

$$\downarrow \text{FixEntry}(i) : \neg \text{FixEntry}(i1), i := i1 + 1, \quad i \leq S, \neg \text{NodeIntvl}('nil', self). \quad (4.15)$$

$$\downarrow \text{Lookup}(self, self, k) : \neg \text{FixEntry}(i), x := self, \quad k := x + \text{pow}(2, i - 1). \quad (4.16)$$

$$\downarrow \text{Finger}(k, s) : \neg \text{LookupRes}(self, k, s), \quad \neg \text{Finger}(k, _), k \neq self. \quad (4.17)$$

$$\downarrow \text{Finger}(k, s) : \neg \text{LookupRes}(self, k, s), \quad \text{Finger}(k, _), k \neq self. \quad (4.18)$$

If the lookup did not succeed at node β , this latter checks its finger table, fetches the best predecessor of node α (highest node Id less than α), and unicasts the *Lookup* request to best predecessor, Rule (4.20) and (4.21), if α 's Id is greater than β 's successors. Otherwise, the lookup request is sent to the successor of node β , Rule (4.19).

4.4 Data centric overlays

$$\begin{aligned} \uparrow \text{Lookup}(@s, x, k) : & \neg \text{Lookup}(\text{self}, x, k), \\ & \text{NodeIntvl}(p, s), k < \text{self}. \end{aligned} \quad (4.19)$$

$$\begin{aligned} \downarrow \text{BestPred}(\text{max}(n1), x, k) : & \neg \text{Lookup}(\text{self}, x, k), \\ & \text{NodeIntvl}(p, s), k > s, s > \text{self}, \\ & \text{Finger}(k1, n1), k > n1. \end{aligned} \quad (4.20)$$

$$\uparrow \text{Lookup}(@n, x, k) : \neg \text{BestPred}(n, x, k). \quad (4.21)$$

An optimization in time and message complexity can be easily done. If the Id of α is less than β 's Id, the lookup request can be sent to the predecessor of β instead of going in the clockwise direction through a very long route. We just replace the attribute "s" in the head of Rule (4.19) by "p".

Each node in the Chord ring holds a list of successors. After joining the ring, each node periodically stabilizes the set of successor list, and the predecessor. Each node checks its successor and calculates the best successor (minimum node Id greater than self) from its *successor list* when its successor leaves the network. Periodically in Rule (4.22), each node say γ asks its successor say λ about λ 's predecessor and successor list, if λ exists on the Chord ring. Otherwise, node γ calculates the best successor using its successor list, and then updates, Rule (4.23), and notifies, (4.24), its new successor.

Upon receiving the successor list from λ , node γ deletes previous successor list and inserts new received ones. Based on the set of successors obtained from stabilization, the best successor is calculated, and the successors that are no longer required are deleted.

$$\begin{aligned} \uparrow \text{AskSucc}(@s, \text{self}) : & \neg \text{NodeIntvl}(p, s), \\ & s <> \text{self}, \text{Route}(s, _). \end{aligned} \quad (4.22)$$

$$\begin{aligned} \downarrow \text{NodeIntvl}(p, s1) : & \neg \text{BestDist}(d), \\ & !\text{NodeIntvl}(p, s), \neg \text{Route}(s, _), \\ & \text{SuccList}(s1), d == s1 - \text{self}. \end{aligned} \quad (4.23)$$

$$\begin{aligned} \uparrow \text{Notify}(@s1, \text{self}) : & \neg \text{BestDist}(d), \\ & !\text{NodeIntvl}(p, s), \neg \text{Route}(s, _), \\ & \text{SuccList}(s1), d == s1 - \text{self}. \end{aligned} \quad (4.24)$$

Upon receiving a notify event by a node, say δ , it checks and updates its predecessor if the predecessor is either 'nil', Rule (4.25), or greater than old predecessor, Rule (4.26). Additional cases such as when the predecessor fails, etc., are also taken into consideration and found in [149].

$$\begin{aligned} \downarrow \text{NodeIntvl}(x, s) : & \neg \text{Notify}(\text{self}, x), \\ & !\text{NodeIntvl}('nil', s). \end{aligned} \quad (4.25)$$

$$\begin{aligned} \downarrow \text{NodeIntvl}(x, s) : & \neg \text{Notify}(\text{self}, x), x < \text{self}, \\ & !\text{NodeIntvl}(p, s), x > p. \end{aligned} \quad (4.26)$$

4.4 Data centric overlays

Ring protocol

The following program, Rules (4.27 - 4.32), defines a circular order on the node Ids (with the smallest node Id, immediate successor of the biggest one), used in the DHT. The index values of each relation are mapped to keys (hash values). Each node is responsible for an interval of these keys, according to the value of its identifier, and their predecessor and successor in the ring.

Since we consider mobile ad hoc networks, the next program relies on a routing protocol, ensuring that each node has routes to all nodes in the network, which are maintained over time. Any participating node may be part of the DHT and hold data related to the application. The program uses the relations *Intvl* with two attributes to save the *predecessor* and the *successor* of each node.

$$\downarrow \text{Intvl}(\text{self}, \text{self}) : - \text{TimeEvent}('Ini'). \quad (4.27)$$

$$\downarrow \text{Extreme}(\min(d), \max(d)) : - \text{TimeEvent}('chNgb'), \text{Route}(d, _). \quad (4.28)$$

$$\downarrow \text{NIntvl}(\max(d1), \min(d)) : - !\text{Extreme}(_, _), \text{Route}(d, _), \\ d > \text{self}, \text{Route}(d1, _), d1 < \text{self}. \quad (4.29)$$

$$\downarrow \text{NIntvl}(y, \min(d)) : - !\text{Extreme}(\text{self}, y), \text{Route}(d, _), d > \text{self}. \quad (4.30)$$

$$\downarrow \text{NIntvl}(\max(d), x) : - !\text{Extreme}(x, \text{self}), \text{Route}(d, _), d < \text{self}. \quad (4.31)$$

$$\downarrow \text{Intvl}(np, ns) : - !\text{Intvl}(_, _), !\text{NIntvl}(np, ns). \quad (4.32)$$

Schema	Description
Intvl(p,s)	Intvl(predecessor, successor)
Extreme(x,y)	Extreme(minId, maxId)
Route(d,z)	Extreme(destination, nextHop)

Table 4.2: Schemas of the *ring* protocol

Each node in the network initially sets its successor and predecessor to its node Id when the program starts with rule (4.27), triggered by the timer "*Ini*" only once. Periodically using the timer "*chNgb*", Rule (4.28), extreme nodes are calculated using the routing table. The third step consists at finding the interval of each node. Rule (4.29) defines new intervals of intermediate nodes by checking the routing table and finding the new predecessor and successor, while Rule (4.30) and (4.31) are used to calculate the interval of extreme nodes. The minimum node, in Rule (4.30), sets its predecessor to the last node and finds its successor, and the maximum node in Rule (4.31) sets its successor to the first node and finds its predecessor. Finally, in Rule (4.32), each node updates its interval upon receiving a new interval.

The virtual ring can be seen on the Figure 4.3. The edges in black are the network edges, while red edges form the virtual ring.

4.4 Data centric overlays

	Table Interval	
Node	Predecessor	Successor
0000	0009	0001
0001	0000	0002
0002	0001	0003
0003	0002	0004
0004	0003	0005
0005	0004	0006
0006	0005	0007
0007	0006	0008
0008	0007	0009
0009	0008	0000

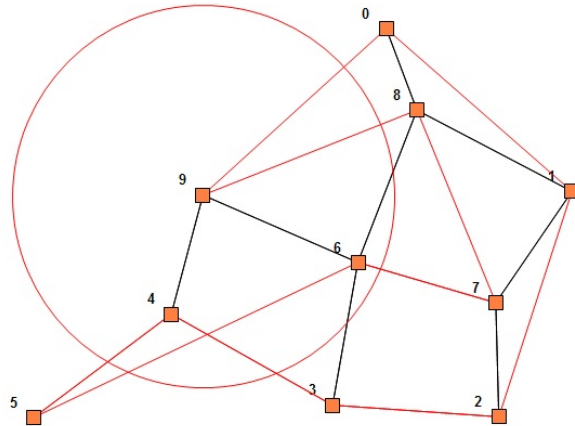


Figure 4.3: A virtual ring

4.4.2 Data replication

In this section, we present two replication protocols. They are related to the distributed lookup protocols presented in Section 4.4.1. In particular to prevent the loss of data and to provide reliability and fault tolerance, we first extend Chord to replicate application data, and second implement a replication protocol that works together with the *ring* protocol. As we said in Section 4.2.1, we suppose that nodes do not leave the network while in processing rounds. Each node therefore handles the received messages, and produces and sends the messages derived in the current round before stopping activity.

Extension of Chord

The Chord protocol is a mechanism for key lookup which maintains a ring of peers. Given a key, it finds the node responsible for it in the ring. Data can be distributed using Chord, by associating to each data item a key, and storing the data on the node responsible for that key. A node say α is responsible for the intervals of keys $[p_\alpha, \alpha]$ where p is the *predecessor* of node α . When node α leaves the network, all the data that have keys in the interval $[p_\alpha, \alpha]$ will be lost. To prevent such loss, we propose a protocol which extends Chord to support the replication of data.

Each peer in the Chord ring maintains a list of r immediate successors. We designed a mechanism that allows each node to replicate its data in its k immediate successors, using its successor list. A node dynamically replicates received data upon their reception, as well a node dynamically recovers stored data upon node failure detection. A failed node is detected by using the routing table without any additional network traffic.

Our discussion in the following program, Rules (4.33 - 4.39), is based on a single replica ($k = 1$) in the best successor, but can be extended easily to multiple replicas (in the r successor list) to cope with higher failure frequencies, at the cost of increased bandwidth usage.

4.4 Data centric overlays

$$\downarrow OldPred(p) : - !Notify(self, x), NodeIntvl(p, s). \quad (4.33)$$

$$\uparrow Replica(@s, fn, dta) : - !DistData(self, fn, dta), NodeIntvl(p, s). \quad (4.34)$$

$$\downarrow TheData(fn, dta) : - !Replica(self, fn, dta). \quad (4.35)$$

$$\begin{aligned} \uparrow Replica(@p, fn, dta) : - !OldPred(op), op <> p, \\ NodeIntvl(p, s), !TheData(fn, dta), \\ k := hash(fn), k \leq op, Route(op, _). \end{aligned} \quad (4.36)$$

$$\begin{aligned} \uparrow Replica(@p, fn, dta) : - !OldPred(op), op <> p, k > op, \\ NodeIntvl(p, s), TheData(fn, dta), \\ k := hash(fn), k \leq p, Route(op, _). \end{aligned} \quad (4.37)$$

$$\begin{aligned} \uparrow Replica(@s, fn, dta) : - !OldPred(op), op <> p, \\ NodeIntvl(p, s), TheData(fn, dta), k > p, \\ k := hash(fn), k \leq op, \neg Route(op, _). \end{aligned} \quad (4.38)$$

$$\begin{aligned} \uparrow Replica(@p, fn, dta) : - !OldPred(op), op <> p, \\ NodeIntvl(p, s), TheData(fn, dta), \\ k := hash(fn), k \leq p, \neg Route(op, _). \end{aligned} \quad (4.39)$$

Schema	Description
OldPred(p)	OldPred(predecessor)
Replica(x,y,d)	Replica(successor, fileName, data)
DistData(x,z,d)	DistData(nodeId, fileName, data)
TheData(z,d)	TheData(fileName, data)

Table 4.3: Schemas of the *Chord extension* for data replication

A peer upon receiving data, Rule (4.34), sends a backup copy to its best successor, which stores the copy upon reception, Rule (4.35). We have seen in the Chord protocol that a node, say α , updates its predecessor upon receiving a notify from other nodes. In the replication program, each node checks its predecessor to delete or replicate data according to the changes of the predecessor. In particular, upon receiving a notify, Rule (4.33), node α saves its old predecessor in the relation *OldPred*. This is used in order to check if old predecessor leaves the chord ring, or a new predecessor joins the chord ring. In Rule (4.36), a new predecessor for node α joins the chord ring. Then, node α (i) sends to new predecessor which will hold the copy, all data that have keys less or equal to old predecessor op , and (ii) delete these data from local data store. At the same time in Rule (4.37), node α sends to new predecessor p all keys that are less than or equal to new predecessor p , which is the new responsible node. In Rule (4.38), node α 's predecessor leaves the network, then node α saves a backup copy on its successor for all data that have keys greater than the new predecessor and less or equal to old predecessor. Moreover, node α sends to new predecessor all the data that have keys less or equal to new predecessor p , Rule (4.39).

Replication on ring

We next present another program for data replication to prevent the loss of data. For reliability purposes, we assume that all data are replicated on two nodes, thus allowing any node to leave the

4.4 Data centric overlays

network with no perturbation to application. This program is used together with the ring protocol to run our online multiplayer game, shown in Section 4.1.3. We assume that each node, say α , is responsible for the interval of values, $[p_\alpha, s_\alpha[$ where p_α and s_α are the predecessor and the successor of α , respectively. The following program, Rules (4.40 - 4.46), ensures the replication of the data on the two nodes responsible for a fragment. It is shown here on the relation *Owner* used in our game example. This relation has three attributes *Avatar*, *Auth* and *Client*, where *Avatar* is the index attribute.

$$\begin{aligned} \uparrow OwnerUpd(@s, x, y, z) : & - TimeEvent('upd'), hv := hash(x), \\ & hv \geq self, Intvl(p, s), Owner(x, y, z), hv < s. \end{aligned} \quad (4.40)$$

$$\begin{aligned} \uparrow OwnerUpd(@s, x, y, z) : & - TimeEvent('upd'), hv := hash(x), \\ & hv \geq self, Intvl(p, s), Owner(x, y, z), s < self. \end{aligned} \quad (4.41)$$

$$\begin{aligned} : & - Intvl(op, os), hv := hash(x), hv < self, \\ & !Owner(x, _, _), !NIntvl(np, ns), np > op. \end{aligned} \quad (4.42)$$

$$\begin{aligned} \uparrow Recover(@np, x, y, z) : & - Intvl(op, _), !NIntvl(np, _), Owner(x, y, z), \\ & hv := hash(x), hv < self, np < op. \end{aligned} \quad (4.43)$$

$$\downarrow Owner(x, y, z) : - !OwnerUpd(self, x, y, z). \quad (4.44)$$

$$\begin{aligned} : & - !OwnerUpd(self, x, y, z), !Owner(r, s, _) \\ & hv := hash(r), hv < self. \end{aligned} \quad (4.45)$$

$$\downarrow Owner(x, y, z) : - Recover(self, x, y, z). \quad (4.46)$$

Schema	Description
Owner(x,y,z)	Owner(avatar, authentication, client)
OwnerUpd(s,x,y,z)	OwnerUpd(successor, avatar, authentication, client)
Recover(p,x,y,z)	OwnerUpd(predecessor, avatar, authentication, client)

Table 4.4: Schemas of the data replication protocol

Periodically using the timer 'upd', each node sends a backup copy to be saved on its successor s for all entries that have index values in the interval $[self, s[$, Rules (4.40). The last node of the ring is managed by rule (4.41). These rules together, on a node say α , prevent the replication of the backup copy of α 's predecessor.

Rules (4.42) and (4.43) are used to manage the backup copy on a node, say β , upon the detection of any change in β 's predecessor. Indeed, if the new predecessor np of node β is greater than the old one op , all entries that have keys (hash values) less than self, β , are deleted, Rule (4.42). However, if the new predecessor np is less than old one op , that means the old predecessor leaves the network, and so all entries that have keys less than β is sent to the new predecessor np , Rule (4.43).

The backup and recover copy are sent using facts of the form *OwnerUpd(nodeId, avatar, owner)* and *Recover(nodeId, avatar, owner)*. The successor saves the backup copy upon receiving the fact *OwnerUpd(self, x, y)* in Rule (4.44) and at the same time deletes the old copy, Rule (4.45), while new predecessor saves a recover copy upon receiving the fact *Recover(self, x, y)* in Rule (4.46).

In the next example, we monitor the impact of the departure of node 7 from the network. Node 7

4.4 Data centric overlays

is in charge of fragments of relations whose index attribute is mapped to a key in the interval $[6, 8[$. The values of the index attributes *Bob* and *Alice* are mapped by the hashing function respectively to node 7, and node 3. Therefore, node 7 holds the fragment of data of the relation *Owner*, with index value *Bob* as seen in Table 4.5. When node 7 disappears, its predecessor node 6 takes the duty, and hosts the fragment, as shown in Table 4.6. Figure 4.4 and 4.5 shows the virtual ring obtained before and after the departure of node 7.

	Table Owner		
Node	Avatar	Auth	Client
0007	Bob	37	0009
0008	Bob	37	0009
0003	Alice	54	0005
0004	Alice	54	0005

Table 4.5: Global view of data before the departure of node 7

	Table Owner		
Node	Avatar	Auth	Client
0006	Bob	37	0009
0008	Bob	37	0009
0003	Alice	54	0005
0004	Alice	54	0005

Table 4.6: Global view of data after the departure of node 7

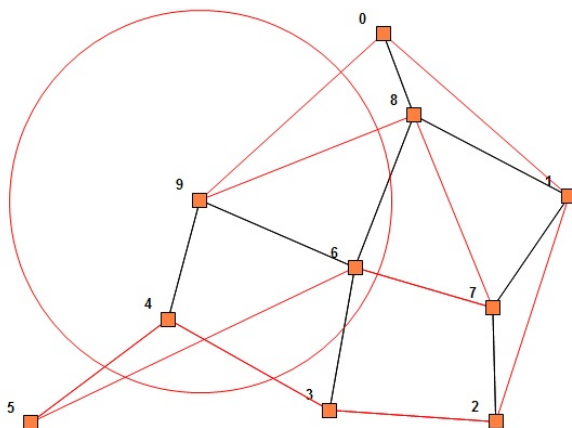


Figure 4.4: Ring before the departure of node 7

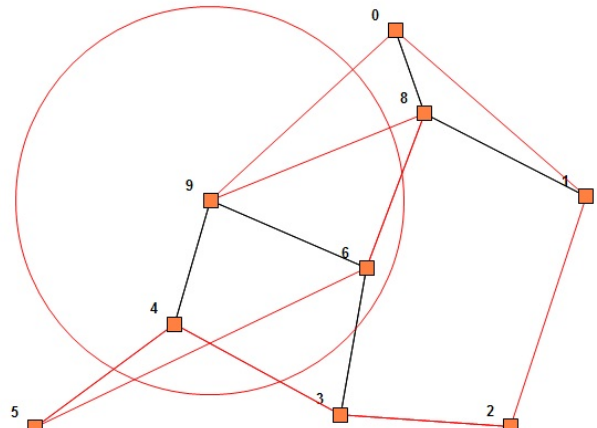


Figure 4.5: Ring after the departure of node 7

4.4.3 Routing

The following program, Rules (4.47 - 4.53), is a simplified version of the DSDV protocol [126], which constructs and maintains proactively all possible routes in an ad hoc networks. We chose DSDV to support mobile clients participating to a multiplayer game in a device to device network. The routes are stored in relation *Route*, described in Table 4.7.

4.4 Data centric overlays

$$\downarrow \text{Route}(\text{self}, \text{self}, 0, 1, 0) : - \text{TimeEvent}('ini'). \quad (4.47)$$

$$\uparrow \text{HelloRt}(\text{self}, x, n, s) : - \text{TimeEvent}('hello'), \text{Route}(x, y, n, s, _). \quad (4.48)$$

$$\downarrow \text{Route}(\text{self}, \text{self}, 0, s, 0) : - \text{TimeEvent}('hello'), \\ !\text{Route}(\text{self}, \text{self}, 0, s', 0), s := s' + 1. \quad (4.49)$$

$$\downarrow \text{Route}(x, \diamond y, n, s, t) : - \text{HelloRt}(y, x, n', s), \\ \neg \text{Route}(x, _, _, _), n := n' + 1, \\ t := m_time + m_timeout. \quad (4.50)$$

$$\downarrow \text{Route}(x, \diamond y, n, s, t) : - \text{HelloRt}(y, x, n', s), s' < s, \\ !\text{Route}(x, y', n'', s'), n := n' + 1, \\ t := m_time + m_timeout. \quad (4.51)$$

$$\downarrow \text{Route}(x, \diamond y, n, s, t) : - \text{HelloRt}(y, x, n', s), n := n' + 1, \\ !\text{Route}(x, y', n'', s), n'' > n' + 1, \\ t := m_time + m_timeout. \quad (4.52)$$

$$: - !\text{TimeEvent}('chkRt'), \\ !\text{Route}(_, _, _, t), t < m_time, t < > 0. \quad (4.53)$$

Schema	Description
Route(d,x,n,q,t)	Route(dest, nextHop, numberHop, sequenceNb, timeout)
HelloRt(s,d,n,q)	HelloRt(sender, destination, numberHop, sequenceNb)

Table 4.7: Schemas of the *DSDV-like* routing protocol

Each node initially creates a route to itself when the program starts with Rule (4.47), triggered by the timer *ini*, which is fired only once. Periodically, using the timer *hello*, each node in Rule (4.48) broadcasts all its route information to its neighbors, and increases the value of the sequence number *destSN*, of the route to itself, using Rule (4.49).

The route information are sent using facts of the form *HelloRt*. A node updates its routing table according to the received route information from its neighbors as follows: (i) a new route is stored Rule (4.50) if there is no route to the same destination in the local route table, (ii) the old route is deleted and replaced with a new one, if the new route has a larger destination sequence number, Rule (4.51), or the new route has the same sequence number as the old one but has a smaller number of hops, Rule (4.52). Each node sets for each route a timeout *m_timeout*, in Rules (4.50), (4.51) and (4.52) upon saving it in the routing table. In Rule (4.53), each node periodically using the timer *checkRoute* deletes all expired routes.

In the following, we show an example of dynamic network in which node 9 is moving between nodes. We monitor and display the content of the routing table of node 9. Figure 4.6 and Table 4.8 represent the network and the content of the table of node 9 before changing the position, while Figure 4.7 and Table 4.9 represent node 9 after changing its position. The route between source node 9 and the destination node 1 is colored in red. DSDV builds routes to all nodes in a network as seen in Table 4.8. DSDV keeps up to date all routes. In particular, it updates periodically all routes. A route may change dynamically with respect to any change in the network. As we notice, the route to destination 1 is changed as seen in Table 4.8 and 4.9.

4.5 A Distributed Server for a multiplayer game

Table route: @node 9				
dest	nextHop	nbhops	routesn	exptime
0000	0008	2	32	103055929
0001	0008	3	30	103055929
0002	0008	3	30	103055929
0003	0008	2	32	103055929
0004	0007	2	32	103057272
0005	0008	2	32	103055929
0006	0006	1	34	103056834
0007	0007	1	34	103057272
0008	0008	1	34	103055929
0009	0009	0	36	0

Table 4.8: Monitoring route to destination 1 before moving node 9

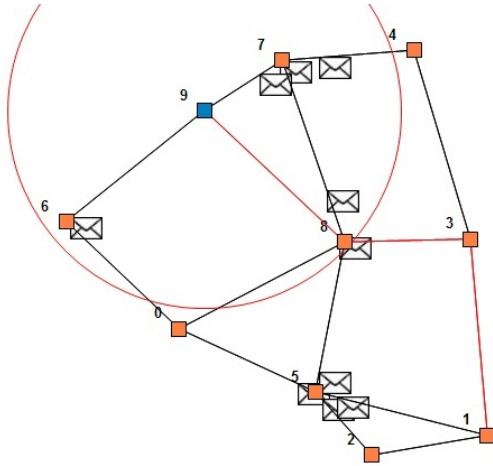


Figure 4.6: Network before moving node 9

Table route: @node 9				
dest	nextHop	nbhops	routesn	exptime
0000	0000	1	54	103190702
0001	0005	2	52	103189468
0002	0005	2	52	103189468
0003	0000	3	50	103190702
0004	0000	4	48	103190702
0005	0005	1	54	103189468
0006	0000	2	52	103190702
0007	0000	3	50	103190702
0008	0000	2	52	103190702
0009	0009	0	56	0

Table 4.9: Monitoring route to destination 1 after moving node 9

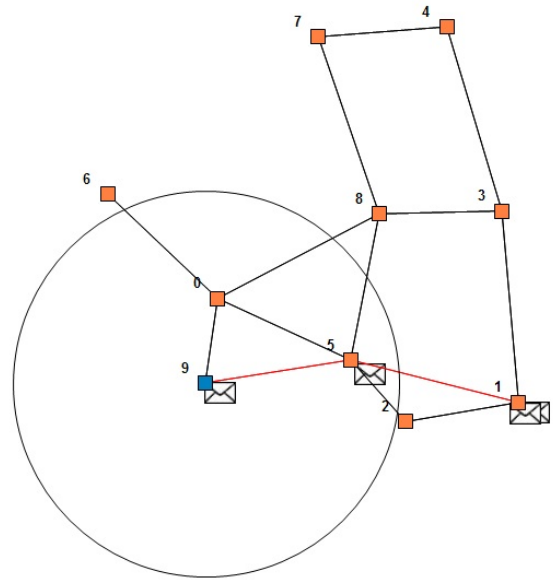


Figure 4.7: Network after moving node 9

4.5 A Distributed Server for a multiplayer game

We have implemented the auction game in Netlog over the QuestMonitor system [29], and made experiments on dynamic networks of several dozen peers participating to the game. We describe below the execution of a simple scenario, where a node joins the network to participate to the game. After retrieving all the information about its avatar, it does a bid on an auction and finally disconnects from the network. At the expiration of the auction, the avatar owns the object, although the peer to whom the avatar belongs has already left the game.

To join the game, a node, say α , goes through four important steps:

- Authentication: Node α sends an authentication message to its neighbors. When the neighbors receive an authentication request, they verify if the authentication key of the avatar is correct, by sending the query to the node which stores the corresponding fragment of the *Owner* table. If the authentication succeeds, the routing algorithm is allowed to start.
- Route propagation: If node α is allowed to join the game, neighbors propagate their routes to node α and accept incoming route from it. The detailed workflow of DSDV is described in

4.5 A Distributed Server for a multiplayer game

Section 4.4.3.

- Insertion in the ring: Node α has to insert itself into the DHT ring. The Ring protocol described in Section 4.4.1 is used to make it responsible for the interval $[pred_\alpha, succ_\alpha[$ of keys (hash values), and to update accordingly the nodes $pred_\alpha$ and $succ_\alpha$.
- Replication: The change in the ring also triggers the replication protocol. The predecessor and the successor of node α in the ring send to node α all entries (data about the distributed table) that have keys in the interval $[pred_\alpha, succ_\alpha[$. The new node is now responsible of some fragments of the distributed database.

The process begins as follows when a new node, *node 7*, joins the game, which is already running between players. The communication between nodes is represented in red in Figure 4.8.

After the authentication and the propagation of the route, *node 7* calculates its interval and saves as predecessor *node 6* and successor *node 8*. It then receives all entries of the distributed table that have index values in its interval $[6, 8[$ from both its predecessor and its successor.

Suppose now that *node 6* has data with index values in the interval $]7, 8]$. The predecessor of the index values is the node responsible. Thus, *node 6* communicates with *node 7* to save the corresponding data, and then saves a backup in its successor *node 8*.

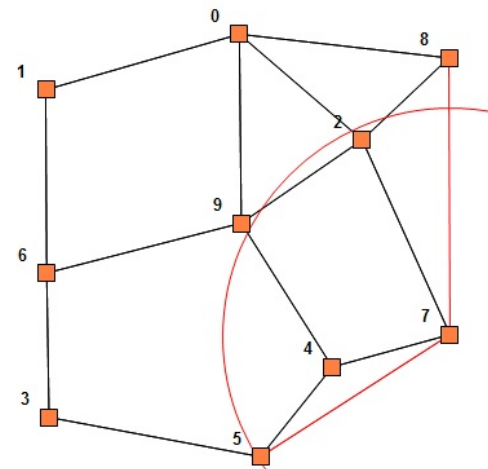


Figure 4.8: Node 7 joins the game

After *node 7* has entered the game, it has to retrieve all the information about its avatar (list of owned objects, bank account, etc.) as well as its views (such as the list of auctions on markets where it registers). It uses the index attributes values to be able to communicate with nodes that are responsible of the fragments of interest to it. When the node has retrieved all the information about its avatar, and has registered to a market, it is able to bid at an auction. It updates its local view of the table *Auction* and then sends an update (client action 3 in Section 4.1.3) to the node responsible of the market. When this node receives the update query, it updates the table *Auction* and the backup copy in its successor (using the replication protocol).

The node can move in the physical space, and so change position in the network. Thanks to the DSDV program, it is always reachable, but the DHT ring is not modified.

Every message sent by a node is composed of an extensional and an intensional destination, as seen in Chapter 3. The extensional destination is the node responsible of the hash values and the intensional destination is a query over relation *KeysIntvl* that can compute a new destination if the extensional destination becomes unreachable.

The DSDV program updates its routes every three seconds. If a node α sends a message to a disconnected node β , a hop in the path between α and β will eventually be missing and the intensional destination will be evaluated to find a new extensional destination. The new destination is the node newly responsible of the hash value. Thus, this ensures persistence of data in the message.

Suppose now that the node disconnects before the end of the auction. The DSDV program detects

4.5 A Distributed Server for a multiplayer game

that the node is no more in the network and the route to this node expires and is deleted. When the route is deleted, the ring, as well as the replication of the data fragments are updated as seen in Section 4.4.2.

When the auction expires, the node responsible of the market where the auction is stored has to execute some system actions. These actions are SQL queries on the distributed tables. They consist of a transaction of SQL queries: (i) update the bank account of the seller and the buyer; (ii) update the owner of the object; and (iii) delete the auction from table *Auction*

To update the bank accounts, the node responsible of the auction, say λ , does a join between the fragments of two tables *Auction* and *Market*, which are on the same node because they have the same index attribute, in order to retrieve the account of the avatar. Then, after hashing the account of each avatar, it sends two messages with extensional/intensional destinations to nodes responsible (say β , γ for the seller and the buyer respectively) in order to update their balance.

Given the restrictions on the queries, messages are never broadcasted, but unicasted thanks to the index attribute of the DHT table. An important feature of our approach, supported by the Nequest machine, is the use of a pair of extensional/intensional destination at the same time, to handle the possibility that the destination node leaves the network.

To update the object, node λ hashes the index attribute value *OID* from its local table *Auction* and sends a message with extensional/intensional destination to node responsible, say θ . Upon receiving the message, node θ updates the entry related to object, *OID*.

Finally, to delete the auction, node λ directly removes the entry of the auction on the local fragment of the auction table, and updates the replicated version.

Conclusion

We have developed a simple programming paradigm for distributed applications, which allows the programmers (i) to write their applications as queries over a client/server architecture, and (ii) to write the underlying overlay network using a data centric language. We have shown that networking protocols could be written as simple, very concise programs consisting of a few dozen of Netlog rules. We described the rules coding for a DHT in a mobile ad hoc network relying on a DSDV like routing protocol. We then showed that applications coded as queries in a client/server framework could be ported seamlessly to the distributed environment. The distributed system based on the DHT ensures the tasks of the centralized server in a fully distributed manner, by relying in the peers which handle horizontal fragments of the relations, and communicate with other peers to solve queries. We have extended the Netquest system, with extensional/intensional destination, which are defined by queries, to provide persistence and resilience. We considered the promising example of multiplayer online games, which can be fully described in a data centric fashion, and showed how it can be seamlessly distributed. The experiments we made on the QuestMonitor platform, with communication based on extensional/intensional destination, demonstrated the robustness of the approach.

The Questlog Language

Contents

Introduction	66
5.1 The Language Questlog	68
5.1.1 The syntax	68
5.1.2 Examples of programs	70
5.2 Procedural Semantics	74
5.2.1 Messages and routing	75
5.2.2 Computation	76
5.2.3 Program execution	78
5.3 Questlog Grammar	80
Conclusion	83

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. In many cases, the provenance of data and services is not relevant, and applications can be optimized by choosing the most efficient solution to obtain data, by minimizing communication for instance.

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

High level programming abstraction, such as data centric programming languages constitute a very promising model for such systems [71]. They allow to specify at a high level "what" to do, rather than "how" to do it. 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.

Declarative networking relies on the rule-based languages [22, 23, 154, 136] developed in the field of databases in the 1980's. It was shown that such languages augmented with communication primitives, allow to express communication protocols or P2P systems with code about two orders of magnitude shorter than imperative programs, and with reasonable execution models. They facilitate not only code reuse among systems, but also the extension, and hybridization.

As we have seen in Chapter 2, different languages have been proposed such as Overlog [96], NDlog [94], Netlog [66], and Webdamlog [4] for high-level programming abstraction. To our knowledge, however, they all follow the *push* mode. They are very successful in expressing various applications and protocols in proactive mode, but less so in reactive mode.

Let us for instance consider the following 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* has been designed to *pull* data from a network by firing a query. *Queries* in *Questlog* are of a very simple form:

$$?R(@x_1, \dots, x_\ell)$$

where R is a *relation symbol* of arity ℓ , and $x_1, \dots, x_\ell$ are variables or constants. The attribute prepended by the symbol @ is an address of a node to which the query will be sent. Queries are associated to rule programs which define their semantics.

Let us illustrate the language on a simplified version of the previous mentioned example. Consider

an application where some sink node collects the positions of nodes which have a temperature greater than some threshold. 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 position. The meaning of the query is defined by a program, installed on each node in the network. This program is used to evaluate the query. Let us consider the following program:

$$\uparrow WarnPos(v, x, y) : - Pos(v, x, y), Tmp(v, t), t > T. \quad (5.1)$$

We assume that each node, say v , stores its location (x, y) as $Pos(v, x, y)$, and its temperature t as $Tmp(v, t)$. When 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 (5.1). 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. Answers need to be sent to the source of the query. This is the role of the affectation operator ($\uparrow$) in front of the rule.

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.

$$\begin{aligned} \uparrow WarnPos(v, x, y) : - & Pos(v, x, y), Tmp(v, t), t > T, \\ & \forall w Link(v, w), ?HighTmp(@w). \end{aligned} \quad (5.2)$$

$$\uparrow HighTmp(v) : - Tmp(v, t), t > T. \quad (5.3)$$

The program is interpreted as follows. The query now matches Rule (5.2). Its body contains facts $Pos(v, x, y); Tmp(v, t)$; as well as an expression:

$$\forall w Link(v, w), ?HighTmp(@w).$$

that contains a new query $?HighTmp(@w)$. The facts are instantiated locally as above. A 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 neighbor β receives the query $?HighTmp(\beta)$. It matches the *head* of rule (5.3). This matching leads to $Tmp(\beta, t), t > T$, the body of Rule (5.3). 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 (5.2) 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 (5.2). Then the result is sent to the initial source of the query $?WarnPos(v, x, y)$.

The Questlog language includes complex primitives such as aggregation, non deterministic choice, etc., to facilitate the programming of complex applications and reactive protocols.

The chapter is organized as follows. In the next section, we present Questlog with the language's primitives and some motivating examples. In Section 5.2, we define the procedural semantics of Questlog, while Section 5.3 is devoted to define the Questlog grammar.

5.1 The Language Questlog

Questlog follows the trend opened by declarative networking [96, 94]. In contrast to Overlog [96], NDlog [94], Netlog [66], and Webdamlog [4], 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. The nodes have initially only the knowledge of their neighbors. The neighborhood relation *Link* is thus distributed over the network such that each node has only a fragment of it.

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 body might be (i) *simple* with no subquery included, it is then evaluated locally on the node, or (ii) *complex* with subqueries, which are sent to the appropriate nodes. Some bookkeeping (BK) is performed to keep track of pending queries and the corresponding subqueries. When the answers are received, the pending query can be computed, and its answer sent to the requesting node.

5.1.1 The syntax

The Questlog language is based on two fundamental objects: (i) a *query* and (ii) a *program*. A program defines the semantics of a query. We first start to present the syntax of a query and their corresponding answer, followed by the syntax of a program. Nevertheless, different aspects that are essential need to be defined first.

A *simple term* is either a variable denoted $x, y, z, \dots$ or a constant which includes *all* for addresses. A *destination term* is of the form $@t$ where t is a simple term. Let us now define the syntax of a query. A *query* is of the form: $?R(@t_1, \dots, t_n)$ where R is a *relation name* of arity n , and the arguments $t_1, \dots, t_n$ are either simple terms or destination term (at most one). The attribute prepended by the symbol $@$ represents the destination of the query. For instance, $?WarnPos(@\nu, x, y)$ is a query where ν, x and y are variables, and $@\nu$ is a destination term. In this case, ν is the destination of the query.

Let us now define the syntax of an answer of a query. An answer is of the form $R(t_1, \dots, t_n)$ where R is a relation name and $t_1, \dots, t_n$ are constants. We refer to an answer interchangeably as a fact. For instance, $WarnPos(\alpha, 4, 8)$ is an answer of the initial query $?WarnPos(@\nu, x, y)$.

A *Questlog program* P is a finite set of rules. Before going in details for the syntax of a rule, we first define some aspects essential for a rule.

A *complex term* is either:

- *aggregate term* of the form $aggr(x)$, where $aggr$ is an aggregate function (min , max , $\#$, or avg) and x is a variable. For instance, $avg(t)$ is an aggregate function where the result is a value that represents the average temperature of t ;
- *random term* of the form $\diamond y$ where y is a variable. It is prepended by the diamond symbol to represent a random function. For instance, suppose that the variable y ranges over α, β and γ . Then $\diamond y$ chooses one random value;
- *arithmetic term* of the form $t_1 \theta t_2$ where t_1, t_2 are *simple* or *arithmetic terms* and θ is an arithmetic function ($+$, $-$, $\times$, $\div$). For instance, $n + 1$ is an arithmetic term.

A *relational head atom* is of the form $R(t_1, \dots, t_n)$ where R is a relation name and $t_1, \dots, t_n$ are either simple terms or complex terms. For instance, $Route(x, \diamond y, z, n)$ is a relational head atom.

5.1 The Language Questlog

A *relational body atom* is of the form $R(t_1, \dots, t_n)$ where R is a relation name and $t_1, \dots, t_n$ are simple terms. For instance, $Route(x, y, z, n)$ is a relational body atom.

A *comparison atom* is an expression of the form $t_1 \theta t_2$, where θ is $=, >, \geq, <, \leq$, or $\neq$ and t_1, t_2 are simple or arithmetic terms. For instance, $n > (k + 1)$ is a comparison atom.

An *assignment atom* is an expression of the form $x := t$ where x is a variable, and t a simple or arithmetic term which does not contain occurrences of x . For instance, $n := k + 1$ is an assignment atom.

A *negative literal* is of the form $\neg F$ where F is a relational body atom. For instance, $\neg R(\alpha, 7, c)$ is a negative literal.

A *universal literal* is of the form $\neg F$ where F is a relational body atom that has unspecified arguments. These arguments have been replaced by the symbol " $_$ ", which is interpreted as universally quantified. For instance $\neg R(\alpha, 7, _)$ is a universal literal, and it is true if for any value c , $\neg R(\alpha, 7, c)$ is true.

A *consumptive literal* is of the form $!F$ where F is a relational body atom. For instance, $!R(x, y)$ is a consumptive literal.

A *forall literal* is of the form $\forall_t F$ where F is a relational body atom, and t a simple term of F . For instance, $\forall_y R(\alpha, y)$ is a forall literal.

A *positive literal* is either a relational body atom, a comparison atom, an assignment atom, or a query atom.

A *literal* is either a positive, a negative, a universal, a consumptive or a forall literal.

Let us now define the syntax of a rule. A rule is of the form:

$$H : - B_1, B_2, \dots, B_n.$$

where $B_1, B_2, \dots, B_n$ is a list of literals, and H is a relational head atom.

In the Questlog language, some primitives are used to represent the storage location and the communication between nodes. We then distinguish between different kinds of rules.

A *store-rule* is an expression of the form:

$$\downarrow H : - B_1, B_2, \dots, B_n.$$

where $\downarrow$ in the head of a rule is the affectation operator used to store results locally. The body contains neither aggregate terms nor random terms. All variables occurring in the head occur also in the body. Variables in the head cannot occur at the same time in distinct types of terms such as simple terms, aggregate terms, or random terms. For instance, $R(x, y, \diamond x, \min(y))$ is not allowed.

A *push-rule* is an expression of the form:

$$\uparrow H : - B_1, B_2, \dots, B_n.$$

where $\uparrow$ in the head of a rule is the affectation operator used to push results to the origin of the query. It satisfies the same conditions as *store-rule*.

A *store-and-push-rule* is an expression of the form:

$$\updownarrow H : - B_1, B_2, \dots, B_n.$$

where $\updownarrow$ in the head of a rule is the affectation operator used to both store results locally and push them to the origin of the query. It satisfies the same conditions as *store-rule*.

5.1 The Language Questlog

A *pull-and-push-rule* is an expression of the form:

$$\uparrow H : - B_1, B_2, \dots, B_n, \overbrace{?R(@x, \dots)}^{\text{optional}}.$$

where $\uparrow$ in the head of a rule is the affectation operator used to push results to the origin of the query. The body contains *at most one rightmost* query atom. This rule satisfies as well the same conditions as *store-rule*.

A *pull-and-store-rule* is an expression of the form:

$$\downarrow H : - B_1, B_2, \dots, B_n, \overbrace{?R(@x, \dots)}^{\text{optional}}.$$

where $\downarrow$ in the head of a rule is the affectation operator used to store results locally. It satisfies the same conditions as *pull-and-push-rule*.

A *pull-and-store-and-push-rule* is an expression of the form:

$$\updownarrow H : - B_1, B_2, \dots, B_n, \overbrace{?R(@x, \dots)}^{\text{optional}}.$$

where $\updownarrow$ in the head of a rule is the affectation operator used to both store results locally and push results to the origin of the query. It satisfies the same conditions as *pull-and-push-rule*.

A *ground rule* is a variable free rule. A ground rule with empty body is an answer.

A *deterministic rule* is a rule without random terms; otherwise it is non-deterministic.

By convention, the name of relations begins with an upper-case letter, while functions, variables, and constants begin with a lower-case letter.

5.1.2 Examples of programs

In this section, we present examples of programs such as *on-demand routing*, *aggregation query*, and *temperature update* expressed in the Questlog language. Here, we consider a high level understanding of programs evaluation, while in Section 5.2, we show in greater details the procedural semantics of programs.

On-demand routing

Let us start with routing which is a fundamental functionality for networks. *On-demand routing* protocols, such as AODV [127], 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.

Consider the query $?Route(\alpha, d, y, n)$, searching a route from node α to destination d with a next hop y , and a length n . The following two rules, Rules (5.4) and (5.5), define an *on-demand routing* protocol, which allows to evaluate the initial query $?Route(\alpha, d, y, n)$.

$$\updownarrow Route(x, w, w, 1) : - Link(x, w). \quad (5.4)$$

$$\updownarrow Route(x, w, z, n+1) : - Link(x, z), ?Route(@z, w, u, n). \quad (5.5)$$

5.1 The Language Questlog

Schema	Description
Link(x,w)	Link(source, destination)
Route(x,w,z,n)	Route(source, destination, nextHop, numberOfHop)

Table 5.1: Schemas of the *on-demand routing* protocol

When node, say α , fires a query $?Route(\alpha, d, y, n)$, node α checks if it matches the head of a rule. In this case, it matches Rules (5.4) and (5.5) which are evaluated in parallel. Local data may satisfy the query, Rule (5.4), which can be answered by $Route(\alpha, d, d, 1)$ (if d is a neighbor of node α). Otherwise, Rule (5.5) generates a body containing a subquery $?Route(@z, d, u, n)$, asking for a route to the destination d , sent to neighbor z . This is the meaning of the "@" symbol in front of variable z in the body of Rule (5.5).

Suppose now that neighbor node, say β , receives the previous query, and that $Link(\beta, d)$ holds on β . The query is evaluated on node β , in a similar fashion. Node β can now run Rule (5.4), and answer the query with $Route(\beta, d, d, 1)$. The result is stored locally on β , due to the affectation operator ($\downarrow$) in front of Rule (5.4), and sent to α , due to the affectation operator ($\uparrow$), where α is the *origin* of the query.

When node α receives the answer $Route(\beta, d, d, 1)$ from node β , it uses again Rule (5.5), but now in push mode as Datalog to derive a fact $Route(\alpha, d, \beta, 2)$ as an answer to the query. As a side effect, intermediate nodes that aggregate answers of subqueries save routes to the destination.

The previous routing program, with Rules (5.4) and (5.5), could lead at the same time to an answer to the query as well as to useless subqueries propagated to other nodes. To prevent propagating subqueries when an answer of a query is found locally, *negation* can be used. Accordingly, the following routing program, Rules (5.6) and (5.7), will be used to evaluate an *on-demand routing* query. Rule (5.7) 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 .

$$\uparrow Route(x, w, w, 1) : - Link(x, w). \quad (5.6)$$

$$\downarrow Route(x, w, z, n + 1) : - \neg Link(x, w), Link(x, z), ?Route(@z, w, u, n). \quad (5.7)$$

When node α fires the query $?Route(\alpha, d, y, n)$, it uses Rules (5.6) and (5.7) to evaluate the query. Rule (5.6) leads to the body $Link(\alpha, d)$. If node d is a neighbor of node α , then Rule (5.6) is satisfied, and Rule (5.7) generates the body:

$$\neg Link(\alpha, d), Link(\alpha, z), ?Route(@z, d, u, n)$$

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. To reduce the delay and the complexity in both communication and computation, an additional rule can be added to benefit from the local knowledge of nodes. The following program with Rules (5.8),

5.1 The Language Questlog

(5.9), and (5.10) defines the semantics of an *on-demand routing* protocol.

$$\uparrow \text{Route}(x, w, \diamond y, n) : - \text{Route}(x, w, y, n). \quad (5.8)$$

$$\downarrow \text{Route}(x, w, w, 1) : - \text{Link}(x, w), \neg \text{Route}(x, w, _, 1). \quad (5.9)$$

$$\begin{aligned} \downarrow \text{Route}(x, w, z, n+1) : & - \neg \text{Link}(x, w), \neg \text{Route}(x, w, _, _), \\ & \text{Link}(x, z), ?\text{Route}(@z, w, u, n). \end{aligned} \quad (5.10)$$

Suppose intermediate node γ has a fact, $\text{Route}(\gamma, d, \theta, 2)$, saved in its routing table. Rule (5.8), when receiving the query $?\text{Route}(\gamma, d, y, n)$, leads to the body $\text{Route}(\gamma, d, y, n)$. The rule is satisfied, then deduced result $\text{Route}(\gamma, d, \theta, 2)$ is sent ($\uparrow$) to the source of the query. In case of plurality of solutions, one route is chosen non-deterministically using the choice operator " $\diamond$ " in front of y . Alternatively, the shortest route could have been chosen using aggregation, (e.g. $\text{Route}(x, w, y, \min(n))$). The evaluation of Rule (5.9) leads to the body $\text{Link}(\gamma, d), \neg \text{Route}(\gamma, d, _, 1)$, where underscore means "any value". The fact " $\neg \text{Route}(\gamma, d, _, 1)$ " is read as follow: there is no route from γ to d with next hop any value and number of hop 1. The use of the negation prevents Rule (5.9) and similarly for Rule (5.10) to be satisfied when a route is found locally. This concludes the *on-demand routing* protocol.

Aggregation query

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 , stores the relation $\text{Tree}(x, y)$ where y is a child of x , and stores a temperature value t in a relation $\text{Tmp}(x, t)$. Suppose node α fires the query, $?\text{ResultAvg}(\alpha, v)$, asking for the average v of the temperature values of deployed sensors in the network. The following program defines its semantics.

$$\downarrow \text{ResultAvg}(x, v) : - v := t/n, ?\text{Avg}(@x, n, t). \quad (5.11)$$

$$\uparrow \text{Avg}(x, 1, t) : - \neg \text{Tree}(x, _), \text{Tmp}(x, t). \quad (5.12)$$

$$\begin{aligned} \uparrow \text{Avg}(x, \Sigma n + 1, \Sigma v + t) : & - \text{Tmp}(x, t), \\ & \forall y \text{Tree}(x, y), ?\text{Avg}(@y, n, v). \end{aligned} \quad (5.13)$$

Schema	Description
ResultAvg(x,v)	ResultAvg(nodeId, temperature)
Avg(x,n,t)	Avg(nodeId, numberOfNodes, temperature)
Tree(x,y)	Tree(nodeId, childId)
Tmp(x,t)	Tmp(nodeId, temperature)

Table 5.2: Schemas of the *aggregation query* program

$\text{Avg}(x, n, t)$ stores the number n of nodes in the subtree rooted at x with the sum t of their temperatures. When node α initially fires the query $?\text{ResultAvg}(\alpha, v)$, node α checks if it matches the head of a rule. The matching leads by Rule (5.11) to the body $v := t/n, ?\text{Avg}(@\alpha, n, t)$ which gives raise to a new query $?\text{Avg}(@\alpha, n, t)$.

The matching of the new query leads either to the body $\neg \text{Tree}(x, _), \text{Tmp}(x, t)$ of Rule (5.12) if α is a leaf (i.e. satisfies $\neg \text{Tree}(\alpha, _)$), or otherwise to $\text{Tmp}(\alpha, t), \forall y \text{Tree}(\alpha, y), ?\text{Avg}(@y, n, v)$

5.1 The Language Questlog

by Rule (5.13). In this later case, a series of queries $?Avg(@y, n, v)$ is 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 (5.12) is satisfied, and the 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 (5.13) 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 (5.13). 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 Σ is the function *sum* and it is used to sum the number of children as well as their temperatures. Node β increases by 1 the number of nodes, adds its temperature to the overall child temperatures, and then sends the result to the source node, its parent in the tree. Rule (5.13) performs a converge-cast of the intermediate results.

When node α receives the answer for the query $?Avg(\alpha, n, t)$, by Rule (5.11), 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*.

Temperature update

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.

$$\downarrow Tmp(x, avg(t)) : - !Tmp(x, t_1), \forall y \text{ Link}(x, y), \quad ?GetNghTmp(@y, t) \quad (5.14)$$

$$\uparrow GetNghTmp(x, t) : - Tmp(x, t). \quad (5.15)$$

Schema	Description
Tmp(x,t)	Tmp(nodeId, temperature)
Link(x,y)	Link(source, destination)
GetNghTmp(y,t)	GetNghTmp(neighborId, temperature)

Table 5.3: Schemas of the *temperature update* program

On each node, say β , the query $?Tmp(\beta, u)$, matches the head of Rule (5.14) thus leading to the body $!Tmp(\beta, t_1), \forall y \text{ Link}(\beta, y), ?GetNghTmp(@y, u)$. It gives raise to queries of the form $?GetNghTmp(@y, u)$ sent to all neighbors y . Upon receiving the query $?GetNghTmp(y, t)$, each neighbor evaluates Rule (5.15) and forwards ($\uparrow$) its own temperature value to the query expeditor β . When all answers (according to $\forall$) are received, Rule (5.14) continues the evaluation in *push* mode, and 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 the local data store. The fact $!Tmp(\beta, t_1)$ is deleted upon evaluating the rule in the *push* mode.

5.2 Procedural Semantics

We consider a message passing model for distributed computation [20], 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 *communication links* between nodes. The nodes have a unique *identifier*, Id , taken from $1, 2, \dots, n$, where n is the number of nodes. Each node has distinct local ports for distinct links incident to it. We make little assumptions on the networks. The communication between nodes rely on messages exchange. The communication is asynchronous. The control is fully distributed in the network, and there is no shared memory.

We have seen in Chapter 3 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;

$$Message = \langle Src, QID, TTL, Payload, Destination \rangle$$

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

$$Payload = \langle Query \mid Answer \rangle$$

The *Destination* 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 Questlog query. The destination has the following format:

$$Destination = \langle ExtDest : IntDest \rangle$$

The architecture of each node is composed of three main components as seen in Figure 5.1.

1. A *Router* to handle the communication with the network;
2. An *Engine* to evaluate the Questlog queries;
3. A Local *Datastore* to manage two sorts of information: all the data of the node, whether related to networking issues (e.g. network topology, routes, etc.) or applications, as well as the rules of the programs.

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:

$$\begin{aligned} Link &= (source, destination) \\ Route &= (source, destination, nexthop) \end{aligned}$$

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. 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.

The Questlog programs are installed on each node in the local data stores, and are used to evaluate Questlog queries fired by the applications or received from other nodes. The evaluation may lead

5.2 Procedural Semantics

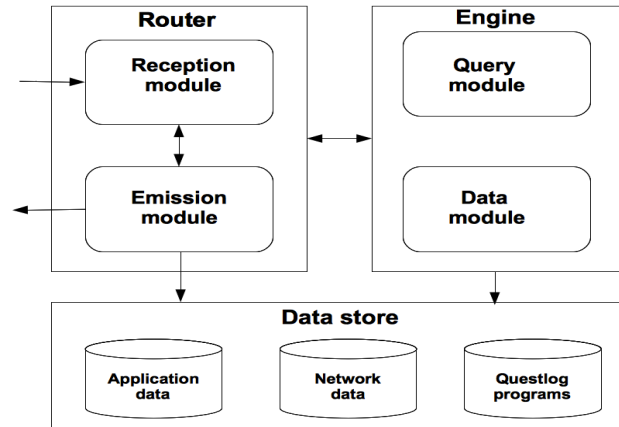


Figure 5.1: The node architecture

to data (as answers) or subqueries sent to other nodes in the network. Pending queries need to be stored, some bookkeeping (BK) 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.

5.2.1 Messages and routing

The Router is used to communicate with the networks. It 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. The Router module on a node α behaves as follows. The router queues the incoming messages on the *reception queue*, $\mathcal{R}^\alpha$, and the messages to send produced by the Engine on the *emission queue*, $\mathcal{P}^\alpha$.

Reception module

The messages on the reception queue, $\mathcal{R}^\alpha$, are sorted according to their destination. For each message M , two cases have to be considered corresponding to the extensional and intensional destinations.

- **Case 1:** Consider the first case where the extensional destination is not empty. Consider the message M with *Destination*:

$$M.Destination = \langle \alpha : query \rangle$$

If the extensional destination is equal to the node address α , or *all*, the node stores the received message with a unique Id and a timeout in a local data structure, *BookKeeping* (*BK*), and the message's contents is transferred to the Engine queue, $\mathcal{L}^\alpha$. Otherwise, the extensional destination is another node address, or *all*. Then the *TTL* is decreased by one. If the new *TTL* value is greater or equal to zero, the message is put on the emission queue, $\mathcal{P}^\alpha$. Otherwise, the message is discarded.

- **Case 2:** Consider now the second case where the extensional destination is empty. Consider the

5.2 Procedural Semantics

message M with *Destination*:

$$M.Destination = \langle - : query \rangle$$

The router evaluates the intensional destination by transferring the *query* to the Engine queue, $\mathcal{L}^\alpha$. The result, received from the Engine and queued in $\mathcal{F}^\alpha$, is a set " σ " of node addresses as follows:

$$\sigma = \{\alpha, \beta, \gamma, \dots\}$$

The router checks if the node address α is in the set. If so, the node stores the received message with a unique Id and a timeout in the local data structure, BK , and the message's contents is transferred to the Engine queue, $\mathcal{L}^\alpha$. Otherwise, the message is discarded.

At the same time, the TTL of the initial message M is updated, and then the message is transferred to the emission module, $\mathcal{P}^\alpha$, 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 M to the emission module, the router could take into consideration the set of received answers σ , encapsulates new messages M_i that have the same contents as initial message M , but with new destinations specified *extensionally* by the addresses in the set σ and *intensionally* by the intensional destination query of M . New messages M_i are transferred to the emission module, $\mathcal{P}^\alpha$. For instance, consider the set of results σ for M 's intensional destination query. The following messages are created and transferred to the emission queue $\mathcal{P}^\alpha$.

$$M_1 = \langle M.contents, \langle \alpha : query \rangle \rangle$$

$$M_2 = \langle M.contents, \langle \beta : query \rangle \rangle$$

$$M_3 = \langle M.contents, \langle \gamma : query \rangle \rangle$$

The important features of this strategy is: (i) toggling from broadcast mode into unicast mode; and (ii) benefiting from local knowledge of a node.

Emission module

The Emission module is used to send messages to other nodes in the network. Each message on the emission queue, $\mathcal{P}^\alpha$, is handled as follows. Either their destination is *all* or *empty*, 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. Other strategies can be implemented as for instance:

- Search for a route to the required destination by firing the query $?Route(\alpha, d, y, n)$ of the *on-demand routing* program (Section 5.1.2);
- Send the message to neighbors;
- Send failure message.

5.2.2 Computation

A message may contain queries (*content-query* or *dest-query*) or data. As we have seen in Chapter 3, *content-query* corresponds to queries in the *payload*, and *dest-query* corresponds to queries in the destination. 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

5.2 Procedural Semantics

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.

The entries in the queue $\mathcal{L}^\alpha$ are treated according to their contents. For each entry, their content is analyzed and transferred to the corresponding module.

Query module

The first step consists in matching the query Q with the *head* of each rule of the corresponding program. Matched rules are loaded from the local data store. 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 the appropriate destinations.

After the evaluation, two cases have to be considered corresponding to the kinds of outputs produced, either (i) a query, or (ii) an answer.

- **Case 1:** The result is a subquery, then the destination to where the subquery should be sent is extracted from the subquery. The destination is the instance of the attribute prepended by the @ symbol. Consider for instance the following subquery $?Route(@\beta, d, u, n)$ which is generated after the evaluation of Rule 5.16. Then, the destination of the subquery is the node address β .

$$\uparrow \text{Route}(\alpha, d, \beta, n+1) : - \underbrace{\text{Link}(\alpha, \beta)}_{\text{local fact}}, \underbrace{?Route(@\beta, d, u, n)}_{\text{subquery to } \beta}. \quad (5.16)$$

The generated subquery needs to be sent to β . A new message is created, having a new query ID, a *TTL*, and α as *Src*. The message is stored in the local data store, BK , and then transferred to the emission module, $\mathcal{P}^\alpha$, of the router.

- **Case 2:** The result is data as answer of the query Q . Different actions are then performed according to the affectation operator of the corresponding rule. The result is stored in the local store due to the affectation operator ($\downarrow$). The result has to be sent to the *origin* of the query due to the affectation operator ($\uparrow$). The result has to be stored and sent due to the affectation operator ($\uparrow$). The result might be as well an answer of the *dest-query*. Two cases have to be considered.
 1. The source of the query Q is a node address. 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 Q is the destination of the message to which the result will be sent. The *QID* of the message should be the same as the query Id of the initial query Q . The corresponding entry that holds these data is retrieved from the local data store, BK . The message is then encapsulated and transferred to the emission module, $\mathcal{P}^\alpha$, of the router.
 2. The source of the query Q is the node address α . The result (an answer of the *dest-query*)

5.2 Procedural Semantics

is put in a set σ , and transferred to the router queue, $\mathcal{F}^\alpha$.

Data module

The data module is used to continue the evaluation of pending queries stored locally in the *BK* on a node. When receiving an answer, the data module first loads the appropriate rules from the local data store, *BK*. More precisely, the engine knows the message *QID* and the other contents of a message, communicated by the router through $\mathcal{L}^\alpha$. The engine matches the received *QID* of the received message with each entry in the *BK* data structure, and retrieves the corresponding Questlog rules. Two cases have to be considered.

- **Case 1:** The rules do not contain forall $\forall$. The engine evaluates the rules in the push mode, as seen in Rule 5.17.

$$\uparrow \underbrace{Route(\alpha, d, \beta, 2)}_{\text{derived fact}} : - \underbrace{Link(\alpha, \beta)}_{\text{local fact}}, \underbrace{Route(\beta, d, d, 1)}_{\text{received fact}}. \quad (5.17)$$

If the body is satisfied, deduced results are sent to their appropriate destination exactly as we have seen previously in the query module (Case 2).

- **Case 2:** The rules contain forall $\forall$. Received answers are saved temporary on the node waiting for all required answers. When all answers are received, the engine evaluates the rules in parallel but now in the push mode. If the body is satisfied, deduced results are again sent to their appropriate destination exactly as we have seen previously in the query module (Case 2).

5.2.3 Program execution

Having define Questlog, we illustrate the execution of the *on-demand routing* protocol via an example of a trivial network shown in Figure 5.2. We suppose node source s fires the query $?Route(s, d, y, n)$ asking for a route to destination d . The variables y and n represent the next hop and the number of hops respectively. The next hop indicates for each route the next hop to route the message in the network. We show that the resulting answer resembles the *on-demand routing* protocol.

In our example, each node is running the *on-demand routing* protocol. We suppose that nodes initially have no intermediate routes on their routing tables. For simplicity, we show only the propagation of queries, the intermediate answers, and the resulting routes.

We show in Figure 5.2 the step-by-step query execution at each node. At Figure 5.2(a), the node s fires the query $?Route(s, d, y, n)$. The engine on node s matches the query with the head of Questlog rules saved in the local data store, and loads only matched rules, Rule (5.8), (5.9), and (5.10) shown in Section 5.1.2. The rules are instantiated by the instances of the variables in the query.

The engine evaluates the rules in parallel on local data. Only Rule (5.10) is satisfied, since there is neither a direct link to the destination d , nor a route, thus leading to a subquery $?Route(@b, d, y, n)$ since b is a neighbor as shown in Figure 5.2(b). The new query $?Route(@b, d, y, n)$ has to be sent to b according to the $@$ symbol.

Similarly, node b loads matched rules and evaluates them leading to subqueries $?Route(@s, d, y, n)$, and $?Route(@c, d, y, n)$, using Rule (5.10), since node s and c are neighbors as shown in Figure 5.2(c). An optimization can be used to avoid subquery to the source of the query. The subquery $?Route(@s, d, y, n)$ can be avoided either by the engine upon evaluation of the initial query (do not

5.2 Procedural Semantics

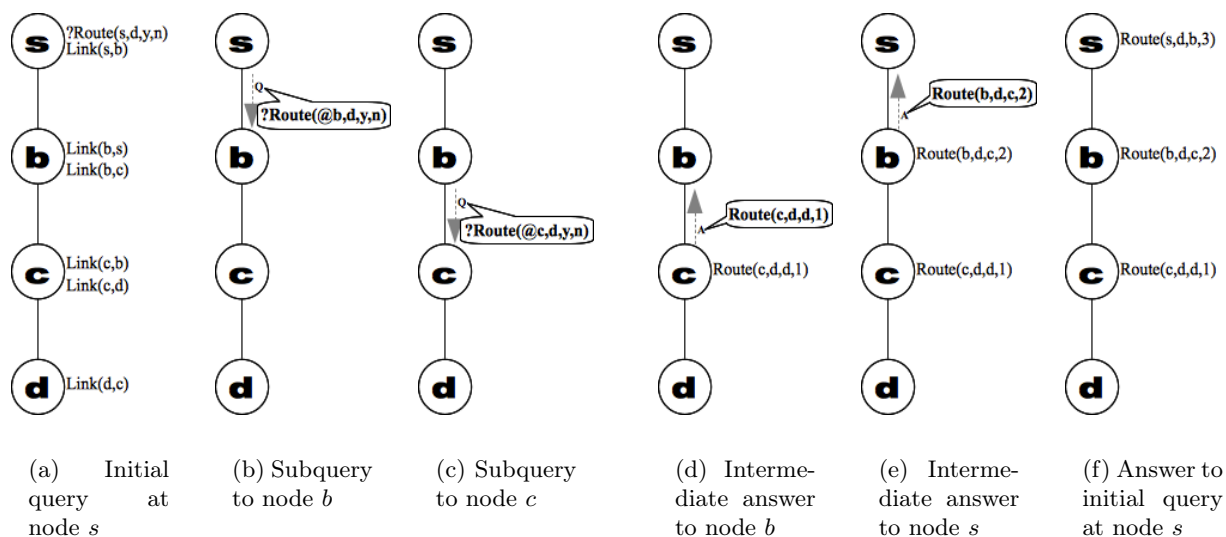


Figure 5.2: Propagation of subqueries and converge-cast of intermediate answers

send subquery to the source of the initial query). Intuitively, this optimization reduces communication overhead. Alternatively, the subquery $?Route(@s, d, y, n)$ can be discarded by the router of node s .

The engine on node c loads matched rules and evaluates them in a similar fashion. However, the evaluation leads to $Route(c, d, d, 1)$ (the head of Rule (5.9)) as an answer of the query since the destination d is a neighbor as shown in Figure 5.2(d). A route is built on node c . It is saved ($\downarrow$) in the local data store and sent ($\uparrow$) to the source node of the query, b . The engine determines the source node by retrieving the appropriate entry from the BK data structure in the local data store.

When receiving the fact $Route(c, d, d, 1)$, the engine on node b matches the QId with the query Id from BK and loads the corresponding rule, Rule (5.10), which is evaluated in the *push* mode. The evaluation leads to a new route $Route(b, d, c, 2)$ to the destination d with an increased number of hop, and having node c as next hop. The new route is saved locally on b and sent to the source node s , as seen in Figure 5.2(e).

Similarly, when receiving the fact $Route(b, d, c, 2)$ on node s , the engine loads the corresponding rule from BK , and evaluates it in the *push* mode. The evaluation leads to new route, $Route(s, d, b, 3)$, to the destination d with next hop node b and number of hop equivalent to 3, saved on node s , as seen in Figure 5.2(f).

Algorithm 5.1 shows the pseudocode of executing rules, Rule (5.9) and (5.10), from the perspective of a single node.

5.3 Questlog Grammar

```

input: a query ?Route(self,d,y,n) or a fact Route(y,d,z,n)

1 switch input do
2   case query ?Route(self,d,y,n) Rules (5.9) and (5.10)
3     if ( $\exists$  Link(self,d) &  $\nexists$  Route(self,d,_,1)) then
4       Save Route(self,d,d,1);
5       Send Route(self,d,d,1) to query expeditor;
6     else
7       foreach neighbor Link(self,y) do
8         Send sub-query ?Route(@y,d,z,n) to neighbor y;
9       end
10    end
11  case fact Route(y,d,z,n) Rule (5.10)
12    if ( $\exists$  Link(self,y) &  $\nexists$  Link(self,d) &  $\nexists$  Route(self,d,_,1)) then
13      Save Route(self,d,y,n+1);
14      Send Route(self,d,y,n+1) to query expeditor;
15    end
16 endsw

```

Algorithm 5.1: Pseudocode corresponding to the execution of the *on-demand routing* protocol

5.3 Questlog Grammar

The Questlog language is formally described using the Extended Backus-Naur Form (EBNF) notation which is a formal mathematical way to specify the syntax of a language.

A Backus-Naur Form (BNF) specification is a set of derivation rules, written as:

$$\langle \text{symbol} \rangle ::= \text{"expression"}$$

where $\langle \text{symbol} \rangle$ is a nonterminal, and the "expression" consists of one or more sequences of symbols. More sequences are separated by the vertical bar, '|', indicating a choice, where the whole being a possible substitution for the symbol on the left. Symbols that never appear on a left side are terminals. On the other hand, symbols that appear on a left side are non-terminals and are always enclosed between angle brackets $\langle \rangle$.

EBNF is an extension of BNF where operators frequently found in the "expression" part are:

- Optional items are enclosed in square brackets (e.g. [$\langle \text{item x} \rangle$])
- Items repeating zero or more times are enclosed in curly brackets or suffixed with an asterisk (e.g. $\langle \text{word} \rangle ::= \langle \text{letter} \rangle \{ \langle \text{letter} \rangle \}$)
- Items repeating 1 or more times are followed by a '+'.
- Alternative choices in a production are separated by the '|' symbol (e.g. $\langle \text{alternative-A} \rangle | \langle \text{alternative-B} \rangle$)
- Where items need to be grouped, they are enclosed in simple parentheses
- Epsilon is used to denote more clearly an empty production
- Terminals are strictly enclosed within double ("...") or single ('...') quotation marks. The angle brackets (" $\langle \dots \rangle$ ") for nonterminals can be omitted.
- A terminating character, the semicolon ";", marks the end of a rule.

5.3 Questlog Grammar

Table 5.4 gathers the main notations:

Usage	Notation
Definition	=
Concatenation	,
Termination	;
Separation	
Option	[...]
Repetition	...
Grouping	(...)
Double quotation mark	"..."
Single quotation mark	'...'
Comment	(*...*)
Special sequence	?...?
Exception	-

Table 5.4: EBNF main notations

It is important to notice that the following conventions are used in EBNF:

- Each meta-identifier is written as one or more words joined together by hyphens "-".
- A meta-identifier ending with "-symbol" is the name of a terminal symbol.

In the following, we present the grammar of the Questlog language. To facilitate the writing of Questlog programs, we highlight in Table 5.5 the equivalent symbols specified in the grammar and used when writing programs to be compiled and executed.

Usage	Syntax Notation	Grammar Notation
Communication	$\uparrow$	$\sim$
Storage	$\downarrow$	\$
Storage and communication	$\updownarrow$	&
Choice	$\diamond$	?
Aggregation	e.g. avg(...)	func_avg(...)
Negation	$\neg$	$\sim$
Deletion	!	!
Forall	$\forall$	[all]

Table 5.5: Equivalent notations in Questlog syntax and grammar

```
//_____
//Identifier
//_____
digit ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0"
upper_case ::= "A" | "B" | ... | "Z"
lower_case ::= "a" | "b" | ... | "z"
letter ::= upper_case | lower_case
ident ::= upper_case { letter | digit }
```

5.3 Questlog Grammar

```
meta_ident ::= "meta_" letter { letter | digit }
variable_ident ::= lower_case { letter | digit }
function_ident ::= "func_" { letter | digit }
//_____
//Constants
//_____
string ::= " ' " { letter | digit } " ' "
number ::= digit { digit }
float ::= number [ "." number ] [ "E" [ "-" ] number ]
bool ::= "true" | "false"
constant ::= [ "-" ] number | [ "-" ] float | bool | string
//_____
//Data Types
//_____
type ::= "int" | "float" | "string" | "boolean"
//_____
//Metadata
//_____
metadata_decl ::= type meta_ident "==" constant "."
metadata_bloc ::= "metadata {" [ metadata_decl ]+ "}"
//_____
//Facts Type
//_____
attribute ::= ident ":" type
fact_type_decl ::= ident " ( " attribute { "," attribute } " ) ."
fact_types_bloc ::= "fact_types {" [ fact_decl ]+ "}"
//_____
//Initial facts
//_____
initial_fact ::= ident "(" constant { "," constant } " ) ."
initial_facts_bloc ::= "initial_facts {" [ initial_fact ]+ "}"
//_____
//Rule head
//_____
head_term ::= constant | [ ? ] variable_ident | meta_ident
               | function_ident "(" [ exp { "," exp } ] ")"
head ::= [ & | ^ | $ ] ident "(" head_term { "," head_term } ")"
//_____
//Expression
//_____
exp ::= multiplicative_exp { additive_op multiplicative_exp }
multiplicative_exp ::= unary_exp { multiplicative_op unary_exp }
additive_op ::= "+" | "-"
multiplicative_op ::= "*" | "/" | "%"
unary_exp ::= number | float | bool | string | meta_ident | variable_ident | parentized_exp |
               negative_exp | function_ident "(" [ exp { "," exp } ] ")"
parentized_exp ::= "(" exp ")"
negative_exp ::= "-" unary_exp
```

5.3 Questlog Grammar

```
//-----  
//Rule Body  
//-----  
body ::= { literal { “,” literal } } [ “,” “->” [ “[All]” | “[One]” ] request] “.”  
request ::= ident “(  
    @ (meta_ident | variable_ident) { “,” body_term } “)”  
    | body_term { “,” body_term } “,”  
    @ (meta_ident | variable_ident) { “,” body_term } “)”  
literal ::= ( [ ~ | ! ] atom ) | condition  
atom ::= ident “(” body_term { “,” body_term } “)”  
body_term ::= exp | “_”  
condition ::= exp condition_op exp  
condition_op ::= “==” | “<>” | “>” | “>=” | “<” | “<=” | “:=”  
//-----  
//Rule  
//-----  
rule ::= [ “-” ident “.” ] [ head ] “:-” body  
rule_bloc ::= “rules {” [ rule ]+ “}”  
//-----  
// Protocol  
//-----  
protocol_bloc ::= “protocol (” ident “)” {” [ metadata_bloc ] [function_bloc] [fact_types_bloc ]  
    [ initial_facts_bloc ] rule_bloc “}”
```

Conclusion

In this chapter, we introduced the Questlog language which has been designed to express distributed programs and applications, and allow to *pull* data from a network. We presented the procedural semantics of Questlog, and illustrated through an example the execution of Questlog programs. We finally described the grammar of the language, which will be used to compile and execute Questlog programs.

In the next chapter, we present the Questlog compiler that transforms the Questlog programs into a sort of bytecode that can be smoothly handled, as well as the Questlog system that executes the received Questlog queries with their corresponding answers.

Processing Questlog Programs

Contents

Introduction	85
6.1 Data Structures	86
6.1.1 Program structure	86
6.1.2 Predefined data structures for programs	87
6.1.3 Predefined data structures for networks	89
6.1.4 Predefined data structures for system	90
6.2 Questlog Compiler	92
6.3 System Architecture	99
6.3.1 Router	100
6.3.2 Questlog Engine	102
6.3.3 Application Programming Interface and Code Editor	105
Conclusion	107

Introduction

Having presented the Questlog language, this chapter describes how Questlog programs can be compiled and executed to implement network protocols and distributed applications. The Questlog language expresses programs at a high level specification. To evaluate smoothly Questlog programs, we have compiled the programs, using Questlog compiler, into a sort of bytecode. The Questlog compiler is based on a compiler developed for Netlog [66].

The Questlog compiler is a multi-pass compiler which transforms the Questlog code to an intermediate bytecode. This bytecode is then executed by the Questlog engine. The generated bytecode is a SQL dialect. Some data structures are thus needed when programs are compiled. These data structures are filled when the generated bytecode is executed by the engine. Afterwards, the engine requests the corresponding data structure to retrieve the appropriate data to evaluate Questlog queries.

The predefined data structures concern those used for (i) Questlog programs such as *metadata*, *initial data*, and *Questlog rules*, (ii) network such as *routing* and *neighborhood* tables, and (iii) Questlog engine such as *reception* and *transmission bookKeeping* as well as other bookKeeping tables used to store relevant data for processing.

The compilation of a Questlog program is done in four steps. Using Java Compiler Compiler (JavaCC), the input Questlog program is first parsed and transformed into an abstract syntax tree. Then the syntax tree is browsed by the semantic analysis module and an enhanced intermediate tree is built. After that, an intermediate code is obtained and finally the output SQL queries are generated.

A query is built for each Questlog operator as follows:

- *Operators for query "?" or for push "↑"*: A SQL query of type *select* is generated;
- *Operator for store "↓"*: A SQL query of type *insert* is generated;
- *Operator for delete "!"*: A SQL query of type *update* for optimization reason is generated.

The system which supports the queries together with their corresponding programs extends a virtual machine called Netquest [66, 29] with a *Questlog Engine*, and a new router. The Questlog Engine executes received queries with their corresponding answers based on related Questlog programs stored in the local data store. In addition, the engine maintains data structures to store mainly pending queries. When answers are received, the pending queries are resumed and new answers are sent to their appropriate destinations.

The Router is used to communicate with the network and to manage messages with extensional and intensional destinations. When a node receives a message, two cases have to be considered corresponding to extensional or intensional destinations. If extensional destination is not empty, the router first checks if the node is the destination. If it is the case, the message is transferred to the engine, and otherwise the message is sent to their appropriate destination. If extensional destination is empty, then the intensional destination is evaluated through the engine and the message is transferred to other nodes in the network.

The Chapter is organized as follows. In the next section, we present the required data structures. The Questlog compiler is described in Section 6.2, while Section 6.3 is devoted to the system architecture.

6.1 Data Structures

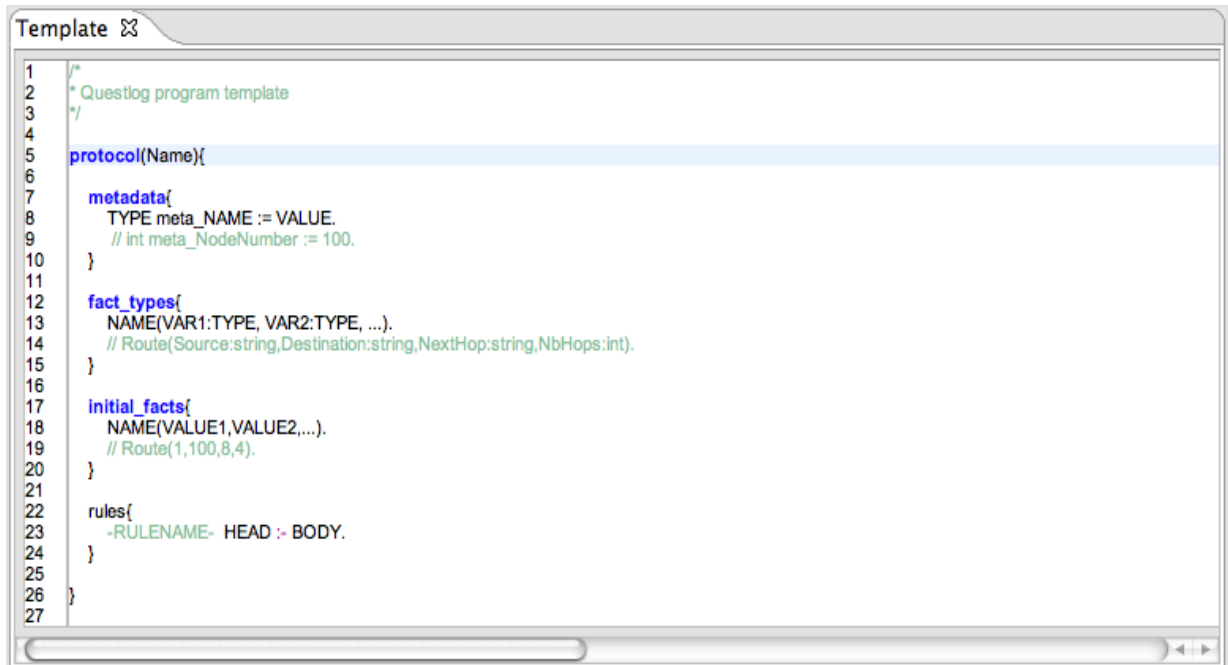
6.1 Data Structures

The Questlog language expresses programs at a high level. To handle smoothly the programs, they are compiled to a SQL dialect that is executed by an engine called *Questlog Engine*. Some data structures are thus needed. These data structures are used by the compiler, filled by the Questlog Engine upon executing the SQL queries resulting from the compilation, and queried by the Questlog Engine to retrieve the appropriate rules to evaluate a Questlog query.

Different concepts are common for most protocols and networks. Examples include *timer*, *meta-data*, *predefined data*, *neighborhood* and *routing* tables, etc. In the following, we first present the structure of a Questlog program, and afterwards we describe the database schemas used either by the compiler, the Questlog Engine or the network.

6.1.1 Program structure

A Questlog program is a set of rules that contain relations with some fixed schema. In Figure 6.1, we show a template for Questlog programs. Schemas used in a program should be declared. We refer to them as *fact_types* as seen in Line (12 – 15). Each program has a name and should be specified as seen in Line (5). In some cases, some programs may have metadata as well as some predefined data. The metadata cannot be modified during the execution of a program. It can be specified as seen in Line (7 – 10). The predefined data are the initial data used when a program runs. They can be deleted or updated during program execution. We refer to predefined data as *initial_facts* as seen in Line (17 – 20). The main rules of programs are specified as seen in Line (22 – 24). For simplicity, we suppose that each rule has a name, Line (23).



6.1 Data Structures

6.1.2 Predefined data structures for programs

Some aspects are essential for programs. It includes a program characterization, metadata specification, and timer. We present in the following the predefined data structures. Underlined attributes are the primary keys.

Protocol

A query is associated with a program (protocol) that defines the semantics of the query. It is used to evaluate the query. Different programs can be installed on nodes of a network, and they could be all active. Table 6.1 gives some characterizations of the program.

<u>PrId</u>	<u>ModuleId</u>	Activated	Mode
varchar	varchar	boolean	varchar

Table 6.1: Program

- PrId: the identifier of the protocol
- ModuleId: the name of the module that contains the Questlog rules
- Activated: a flag to indicate if the program is activated
- Mode: the name of the engine that is used to execute the protocol

Metadata

Metadata is data providing information about some data, stored and managed in a database. In our system, metadata are variables of type *int*, *string*, *float* or *boolean*. These variables have predefined values that cannot be modified at run time by neither a programmer nor a system. As we have seen in Section 6.1.1, the metadata are a part of a program. Each time, the metadata are modified by a programmer, the program should be recompiled and installed on nodes of a network. Table 6.2 defines the metadata data structure with the corresponding attributes.

<u>Name</u>	Type	Value	<u>PrId</u>
varchar	varchar	varchar	varchar

Table 6.2: Metadata

- Name: the name of the metadata
- Value: the value of the metadata
- Type: the type of the metadata.
- PrId: the Identifier of the protocol the metadata belong

Timer

Each program may have a timer. It should be defined in the *initial_facts* part of the program structure presented in Figure 6.1. Table 6.3 shows the required attributes to define a timer. Note that the primary key is an auto-increment identifier.

6.1 Data Structures

Name	Interval	Occurrence	PrId
varchar	int	int	varchar

Table 6.3: Timer

- Name: the name of the timer
- Interval: the period to wait before sending an event. By convention, millisecond is the unit of measurement for time periods.
- Occurrence: the number of time the timer is repeated. By convention, 0 is used for infinite time
- PrId: the identifier of the protocol the timer belongs

Questlog pull rules

The evaluation of Questlog rules follows the *backward* and *forward* chaining mechanism. Rules can be either *simple* or *complex*. Simple rules are in the *pull* mode, while complex rules that have queries in their bodies are at the same time in the *pull* and *push* mode. Relevant informations of pull rules are stored in Table 6.4.

PrId	FactName	RuleName
varchar	varchar	varchar

Table 6.4: Questlog pull rules

- PrId: the identifier of the protocol
- RuleName: the name of the rule
- FactName: the name of the head relation in simple rule. In complex rule, however, it is the name of the query relation in the body.

Questlog push rules

We have seen previously in *Questlog pull rules* that *complex* rules are at the same time in the *pull* and the *push* mode. When an answer is received, the rule is evaluated in the *push* mode. Nevertheless, a particular attention should be taken for rules that contain forall ($\forall$). Relevant informations of *push* rules are stored in Table 6.5.

PrId	FactName	RuleName	ForAll
varchar	varchar	varchar	boolean

Table 6.5: Questlog push rules

- PrId: the identifier of the protocol
- RuleName: the name of the rule
- FactName: the name of the head relation

6.1 Data Structures

- ForAll: a boolean value that is true if a rule contains forall ($\forall$)

Questlog variable mapping

Complex rules contain queries in their bodies. These rules are used when a local data do not satisfy a query. They are used to send subqueries to other nodes in the network and to converge-cast of answers. Each complex rule is rewritten to two rules: (i) the first sub-rule in the pull mode corresponds to the subquery in the body of the initial rule, (ii) the second sub-rule in the push mode corresponds to the answer. When rewriting a rule, a particular attention should be taken for variables used in the *head* as well as in the *body* of a rule. Table 6.6 is used to map variables.

PrId	RuleName	HeadPosition	AttributeName	AttributeType
varchar	varchar	int	varchar	varchar

Table 6.6: Questlog variable mapping

- PrId: the identifier of the protocol
- RuleName: the name of the rule
- HeadPosition: the position of the variable in the head
- AttributeName: the name of the first attribute in the body that corresponds to the variable in the head

6.1.3 Predefined data structures for networks

In Questlog, we suppose that each node has knowledge about their neighbors. In this case, we predefine the relation *Neighbor* that can be used directly without declaration when writing a program. In addition, since queries could be sent to other nodes in a network, a routing table *Route* is predefined.

Neighbor

In the programs presented in Chapter 6, we have used the relation *Link*(*Source*, *Destination*) instead of *Neighbor*(*Destination*) for simplicity. In fact, the attribute *Source* in the relation *Link* represents the node self address. In the implementation, however, we use the relation *Neighbor* that contains only one attribute which represents the neighbors' addresses to prevent data duplication.

Neighbor	Additional attribute
int	type

Table 6.7: Neighborhood table

It is note worthing to mention that the relation *Neighbor* can be extended to have more attributes such as the cost for instance. In this case, these new attributes should be declared when writing a program in the "*fact_types*" part as seen in Figure 6.1.

6.1 Data Structures

Route

The routing table is essential for distributed programs in order to route queries and their corresponding answers to their destinations. To route a message, each node saves next hop on the route to get the required destination. We thus need at least two attributes in the routing table, $Route(Destination, NextHop)$. Additional attributes can be added but they should be declared when writing a program.

Destination	NextHop	Additional attribute
int	int	type

Table 6.8: Routing table

6.1.4 Predefined data structures for system

The system uses predefined data structures to evaluate Questlog programs. Some bookKeeping are performed to keep track of messages as well as queries sent to other nodes in the network. In the following, we describe predefined data structures for system.

Query store

The compiler translates the Questlog rules to SQL queries (Section 6.2). The system evaluates programs by running the corresponding SQL queries, stored in the appropriate data structure. Table 6.9 is used to store the translated SQL queries as well as some other required informations such as the name of the rule, the identifier of the program, the identifier of the query, the fact name, and the name of the attribute which is prepended by a diamond (random) operator, in the head of a rule.

RuleName	<u>PrId</u>	<u>QId</u>	Query	FactName	HeadAtt
varchar	varchar	varchar	varchar	varchar	varchar

Table 6.9: Query store

- RuleName: the name of the rule
- PrId: the identifier of the protocol
- QId: an unique identifier by protocol generated by the compiler
- Query: the SQL query corresponding to the rule
- FactName: the name of the fact in the head of the rule
- HeadAtt: this attribute is used when the rule has diamond operator in the head, and it should be filled with the name of the attribute which prepended by the diamond operator.

Reception bookKeeping

We have seen in Chapter 5 that all received messages are stored in a local data structure. In Table 6.10, we show the reception bookKeeping data structure which is used by the engine to save all

6.1 Data Structures

relevant information in received messages. The payload of messages are stored in separate table shown in Table 6.11.

<u>LocalId</u>	Source	Forwarder	MessageId	PayloadId	PrId
int	varchar	varchar	int	int	varchar

Table 6.10: Reception bookKeeping

- LocalId: a local identifier of the message incremented automatically
- Source: the source node that fires the query received in message
- Forwarder: the node that transfers the query
- MessageId: the identifier of the message
- PayloadId: the identifier of the payload
- PrId: the identifier of the protocol

Payload bookKeeping

With Questlog, nodes fire queries. A message's payload may contain a query or an answer. Table 6.11 is used to store the payload of received messages.

<u>PayloadId</u>	QId	IsQuery	FactName	Attributes	ToTreat
int	varchar	boolean	varchar	varchar	boolean

Table 6.11: Payload bookKeeping

- PayloadId: the identifier of the payload
- QId: the identifier of the query
- IsQuery: a boolean value to represent the content of the payload
- FactName: the name of the fact on the payload
- Attributes: the attributes of the fact in a string format
- ToTreat: a boolean to indicate if the entry has to be treated by the engine.

Transmission bookKeeping

The evaluation of a query in Questlog may give raise to other queries in the network. Pending queries need to be stored. Table 6.12 is used to store relevant data that will be used when answers are received.

Id	Destination	PId	PayloadId
int	varchar	varchar	int

Table 6.12: Transmission bookKeeping

- Id: an identifier of the message

6.2 Questlog Compiler

- Destination: the destination of the message
- PId: the identifier of the protocol
- PayloadId: the identifier of the payload

Intensional destination bookKeeping

We have seen in Chapter 5 that a message may contain *content-query* in the payload and *dest-query* in the destination. The router first transfer the *dest-query* to be evaluated by the engine. The message is pending until getting an answer from the router. A data structure is needed to save temporary the *content-query* in pending messages. Table 6.13 stores the content of the payload of the received message.

PayloadId	FactName	Attributes
int	varchar	varchar

Table 6.13: Intensional destination bookKeeping

- PayloadId: the identifier of the payload
- FactName: the name of the fact on the payload
- Attributes: the attributes of the fact in a string format

Forall bookKeeping

The Questlog rules may contain the primitives $\forall$ that means a node should wait all answers before resuming the evaluation of a pending query. Answers received should be stored. Table 6.14 is used to save received answers.

QueryId	QueryLocalId	Destination	AnswerLocalId
varchar	int	varchar	int

Table 6.14: Control for all answers (ControlAnswersForAll)

- QueryId: the identifier of the query
- QueryLocalId: a localId of the query
- Destination: the destination of the query
- AnswerLocalId: an unique identifier of received answer

6.2 Questlog Compiler

The Questlog compiler is a multi-pass compiler which compiles the Questlog code to an intermediate bytecode. This bytecode is then executed by the Questlog engine. Figure 6.2 shows the architecture of the compiler. The compilation is done in four steps: (i) lexical analysis, (ii) syntax analysis, (iii) semantic analysis, and (iv) code generation.

6.2 Questlog Compiler

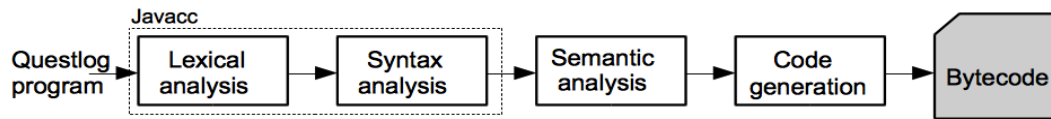


Figure 6.2: Compiler architecture

The input program is parsed and transformed into an abstract syntax tree. Then the syntax tree is browsed by the semantic analysis module and an enhanced intermediate tree is built. Finally, the output bytecode is generated.

Lexical Analysis

The first step is composed of the lexical and the syntax analysis, which constitute what is called a parser. This step is done by using the open source Java Compiler Compiler (Javacc) and the Questlog grammar defined in Extended Backus-Naur Form, as we have seen in Chapter 5.

The lexical analysis breaks the input source code into tokens. Each token is a single atomic unit of the language. It corresponds for instance to keywords, punctuation and literals. The lexical analysis does not take into account white spaces and comments. Javacc has been used to convert the input source code, a Questlog program, into a sequence of tokens.

Syntax Analysis

The syntactical analysis constitutes the most important part of the parser. The goal is to extract the meaning of a program while ensuring the syntactical correctness of the source code. The syntactical analysis output is an internal representation that corresponds to an abstract syntax tree of the source code. During syntactic analysis, the compiler examines the program source code with respect to the grammar of the Questlog language defined in Chapter 5. If a rule is violated, the compiler displays an error message.

Semantic Analysis

The syntax tree obtained from the parser is browsed by the semantic analysis module and an enhanced intermediate tree is built. During this phase, the semantic analysis module performs for each rule semantic checks such as type checking (checking type errors), object binding (associating variable and function references with their definitions), and object assignment (requiring all local variables to be initialized before use). If there is any error at the end of this stage the compilation is stopped, and a rejection message of incorrect program is issued.

Code Generation

The code generation module generates the bytecode. The Questlog programs are transformed into a sort of bytecode that can be smoothly handled. We compile the Questlog programs into a SQL dialect that is executed by the engine. In the following, we present the translation to SQL of the different parts of a program.

6.2 Questlog Compiler

Module

The first step consists on creating a SQL query that fills the table *Program*, Table 6.1. A tuple has to be added for each protocol. Thus a SQL query of type *insert* should be used. The tuple is composed of the protocol's name, the module's name, the corresponding engine, and a flag to activate or not the module.

Listing 6.2 shows the translation of the main structure of program, Listing 6.1, concerning the protocol *OnDR* to a SQL query.

```
protocol(OnDR){  
    ...  
    rules{  
        -R1- HEAD :- BODY.  
    }  
}
```

Listing 6.1: Program with unique module

```
INSERT INTO Protocol (PrId, ModuleId, Activated, Mode) VALUES ('OnDR',  
    'rules', 1, 'Questlog');
```

Listing 6.2: Translation of a program main structure to SQL

Metadata

Metadata should be inserted in table *Metadata*, Table 6.2. A SQL query of type *insert*, Listing 6.4, is used for each predefined metadata, Listing 6.3.

```
metadata{  
    int meta_Number := 5000.  
    string meta_CharSet := 'example'.  
}
```

Listing 6.3: Questlog metadata

```
INSERT INTO metadata (Name, Type, Value, PrId ) VALUES('meta_Number',  
    'int', '5000', 'OnDR');  
  
INSERT INTO metadata (Name, Type, Value, PrId ) VALUES('meta_CharSet',  
    'string', 'example', 'OnDR');
```

Listing 6.4: Translation of Questlog metadata to SQL

Schemas

Schemas are defined in the *fact_types* section of Questlog program structure. For each schema, the corresponding table has to be created. The table is composed of schema's attributes. In Table 6.15, we present the data type conversion between Questlog and SQL.

6.2 Questlog Compiler

Questlog data type	SQL data type
string	varchar(255)
int	int
boolean	tinyint
float	float

Table 6.15: Questlog to SQL data type conversion

Listing 6.5 shows examples of schemas defined in section *fact_types*. It contain a new relation *RelName* with their related attributes, as well as an extension of the predefined table *Route* by declaring new attribute.

```
fact_types{
    RelName(Source:string , Flag:boolean , NumA:int , NumB:float ).
    Route(NbHops:int) .
}
```

Listing 6.5: Questlog schemas

To translate new schema, a SQL query of type *create table* is used. Listing 6.6 shows the creation for the new table *RelName*, taking into consideration the Questlog to SQL data type conversion as shown in Table 6.15. The constraint unique key is particularly chosen to prevent two similar facts to be inserted on the same table.

```
CREATE TABLE RelName(Source VARCHAR(256) NOT NULL, Flag TINYINT(1) NOT
NULL, NumA INT NOT NULL, NumB FLOAT NOT NULL, id INT AUTO_INCREMENT
PRIMARY KEY, CONSTRAINT UNIQUE uk (Source , Flag , NumA, NumB))
```

Listing 6.6: Translation of a schema

The two tables *Route* and *Neighbor* are already predefined in the system as we have seen in Section 6.1.3. Additional attributes should be declared in Questlog. In SQL, we need to alter the corresponding table by adding required attributes and updating the constraint unique key.

```
ALTER TABLE Route ADD NbHops INT , DROP INDEX uk , ADD CONSTRAINT UNIQUE
uk (Dest , NextHop , NbHops);
```

Listing 6.7: Translation of altering predefined schema

Initial facts

Initial facts are defined in the *initial_facts* section of the Questlog program structure. Initial fact are translated to a SQL query of type *insert*. The schema of the corresponding fact has to be defined before in the section *fact_types* of the Questlog program structure. If the schema is not defined, or if the number of attributes as well as the type of attributes are different from the defined schema, then the compilation fails. Listing 6.8 and 6.9 shows the translation of initial fact timer.

6.2 Questlog Compiler

```
initial_facts{
    Timer('Ini',1,1,'OnDR').
}
```

Listing 6.8: Questlog initial facts

```
INSERT INTO Timer(Name, Period, NbOccurrence, PrId) VALUES('Ini', 1,
1, 'OnDR');
```

Listing 6.9: Translation of initial fact timer

Rule

Questlog rules are composed of conditions and actions. The actions (head or sub-query in the body) are performed when all the conditions (body except Questlog query) are satisfied. Similarly, a SQL query is composed of conditions (the WHERE clause) and actions (the SELECT clause including queries of type SELECT, INSERT, DELETE, UPDATE). The principle of the translation from Questlog rules to SQL queries is to translate the conditions of a Questlog rule into the conditions of SQL queries and the actions of a Questlog rule into the actions of SQL queries. Note that all generated SQL queries are stored in the table *query store*, Table 6.9, where each tuple is composed of a SQL query and other required information gathered by the compiler and used by the system for processing (mainly in order to collect and obtain an answer in the Questlog form to be sent to their appropriate destination).

In a rule, a SQL query is built for each Questlog operator; query "?" (SQL query of type SELECT), store "↓" (SQL query of type INSERT), push "↑" (SQL query of type SELECT) and deletion "!" (SQL query of type UPDATE). These operators are decomposed into two parts: (i) push (↑) and store (↓) that occur in the head, and (ii) query "?" and deletion "!" that occur in the body. In the following, we show through examples the translation of Questlog rules using examples that have the following schema: A(a,b), B(a,b), C(a,b) and D(a,b).

We have seen in Chapter 5 that the evaluation of a Questlog rule may lead either to an answer (the deduced head of the rule), or to a subquery (specified in the body of the rule).

- Let us start the translation of a simple Questlog rule that, if satisfied, leads to an answer. Deduced head corresponds to an answer of the query. The deduced head is composed of a fact name as well as instances of its variables. For instance, $A(\alpha, \beta)$ is a fact. The values α and β are instances of the variables a and b that are occurred in the body of a rule. In SQL, these variables should be selected from the body. The compiler reads variables in the body of a rule going from left to right. If a variable appears more than one time in the body, then the compiler selects the first variable found.

Push: Rule (6.1) shows a rule that has the push operator (↑). Thus a SQL query of type SELECT is used. Two positive literals in the body of the rule share a common variable, y . The WHERE clause is composed of a part for each common variable. For variable y , the where clause is $B.b=C.a$. There is no condition for variable x and z because they are not shared between literals. The generated SQL query for Rule (6.1) is shown in Listing 6.10.

$$\uparrow A(x, z) : -B(x, y), C(y, z). \quad (6.1)$$

6.2 Questlog Compiler

```
SELECT B.a , C.b FROM B,C WHERE B.b = C.a ;
```

Listing 6.10: Generated SQL query for Rule (6.1)

Let us now see how to translate a rule that contains a negative literal in its body as shown in Rule (6.2). The negative literal means that a specified tuple do not exist in the corresponding table. For instance, $\neg A(\alpha, \beta)$ is read as follow: there is no tuple (α, β) in the relation A . The negative literal is translated into a NOT EXISTS clause. The generated SQL query for Rule (6.2) is shown in Listing 6.11.

$$\uparrow A(x, z) : \neg B(x, y), C(y, z), \neg A(x, z). \quad (6.2)$$

```
SELECT B.a , C.b FROM B,C WHERE B.b = C.a AND NOT EXISTS (SELECT *
FROM A WHERE A.a = B.a AND A.b = C.b) ;
```

Listing 6.11: Generated SQL query for Rule (6.2)

Let us now see how to translate a rule that contains condition and assignment in its body as shown in Rule (6.3). The allowed condition in Questlog language are $<$, $>$, $<=$, $>=$ and $<>$. Each variable has to be defined in the rule before being used in a condition. For assignment ($:=$), the translation in SQL is $(=)$. Variables on the right part of an assignment operator should be defined before in the body. However, it is not the case for the variable on the left part. If a variable used in the body is not defined neither in the body nor in the head, a warning message is thrown. The generated SQL query for Rule (6.3) is shown in Listing 6.12.

$$\uparrow D(x, y) : \neg A(x, y), x < y, x := 100. \quad (6.3)$$

```
SELECT 100 , A.b FROM A WHERE 100 < A.b ;
```

Listing 6.12: Generated SQL query for Rule (6.3)

Store: The generation of a SQL query for a rule that has an operator store ($\downarrow$) in the head is treated the same as for the push operator. One exception is that the result should be stored locally on the node. Thus a SQL query of type INSERT followed by a query of type SELECT is used. The generated SQL query for Rule (6.4) is shown in Listing 6.13.

$$\downarrow A(x, z) : \neg B(x, y), C(y, z). \quad (6.4)$$

```
INSERT INTO A(a , b)
SELECT B.a , C.b FROM B,C WHERE B.b = C.a ;
```

Listing 6.13: Generated SQL query for Rule (6.4)

Delete: The deletion operator "!" means that the fact that are used in the body is deleted after evaluation of related rule. In the data structures that we have defined and for optimization

6.2 Questlog Compiler

reason, we have used a system attribute *deleted* of type boolean in order to specify if the related entry is to be deleted. Thus, a SQL query of type UPDATE is used. The generated SQL query for the deletion operator "!" in Rule (6.5) is shown in Listing 6.14.

$$\downarrow A(x, z) : \neg !B(x, y), C(y, z). \quad (6.5)$$

```
UPDATE B,C SET B.deleted = true WHERE B.b = C.a;
```

Listing 6.14: Generated SQL query for the deletion operator "!" in Rule (6.5)

Random: A Questlog rule may contain a random function that prepends a variable used in the head of the rule. The result is grouped according to the variables in the head. The generated SQL query for Rule (6.6) is shown in Listing 6.15.

$$\uparrow D(x, \diamond z) : \neg A(x, z), x < y, y := 100. \quad (6.6)$$

```
SELECT random.a, random.b FROM (SELECT A.a AS a, A.b AS b FROM A
WHERE A.a < 100 ORDER BY RAND()) AS random GROUP BY random.a;
```

Listing 6.15: Generated SQL query for Rule (6.6)

- Let us move on to see the translation of a Questlog rule that contains a (sub-)query in its body. The evaluation of such a rule, if satisfied, leads to (sub-)queries to be sent to other nodes in a network. Thus SQL queries of type SELECT are used.

Pull: A Questlog query contains variables and at least one constant that represents the destination. For instance the query $?C(@\beta, z)$ has z as a variable, and β as an instance for the variable y in the table B . If other constants are used in a query, they should be defined in the body or appeared in the head of a rule. Variables that are used in a query may be appeared in the head of a rule or they are new variables.

Rule (6.7) is translated into two SQL queries corresponding to (i) Questlog query operator "?" in the body, and (ii) push operator " $\uparrow$ " (could be as well store operator " $\downarrow$ " or both). The generated SQL query for the query operator "?" in Rule (6.7) is shown in Listing 6.16.

$$\uparrow A(x, z) : \neg B(x, y), ?C(@y, z). \quad (6.7)$$

```
SELECT B.b, A.b FROM A,B;
```

Listing 6.16: Generated SQL query for the query operator "?" in Rule (6.7)

Let us now see how the second SQL query corresponding to push operator " $\uparrow$ " is generated. This is done by ignoring the query operator in the body of Rule (6.7), and evaluating the rule in the push mode when an answer to the (sub-) query is received, as if the case of Rule (6.1). The generated SQL query is the same as the SQL query shown in Listing 6.10.

6.3 System Architecture

Let us take another example of a Questlog rule that contains a negation in its body, as shown in Rule (6.8). This rule is translated into two SQL queries as shown in Listings 6.17 and 6.18.

$$\uparrow A(x, z) : \neg B(x, z), B(x, y), ?C(@y, z). \quad (6.8)$$

```
SELECT B2.b, A.b FROM A,B AS B2 WHERE NOT EXISTS ( SELECT * FROM B
WHERE B.a = B2.a AND B.b = A.b );
```

Listing 6.17: Generated SQL query for the query operator "?" in Rule (6.8)

```
SELECT B1.a, C.b FROM B AS B1,C WHERE B1.b = C.a AND NOT EXISTS (
SELECT * FROM B WHERE B.a = B1.a AND B.b = C.b );
```

Listing 6.18: Generated SQL query for the push operator "↑" in Rule (6.8)

6.3 System Architecture

In this section, we present the system which supports the queries together with their corresponding programs. The network is constituted of nodes that communicate by exchanging messages through communication channels. We make no particular assumption on the devices or the channels. Each node is equipped with an embedded machine to evaluate the application queries and the programs. The Questlog programs are installed on each node of the network, and all the nodes have the same behavior.

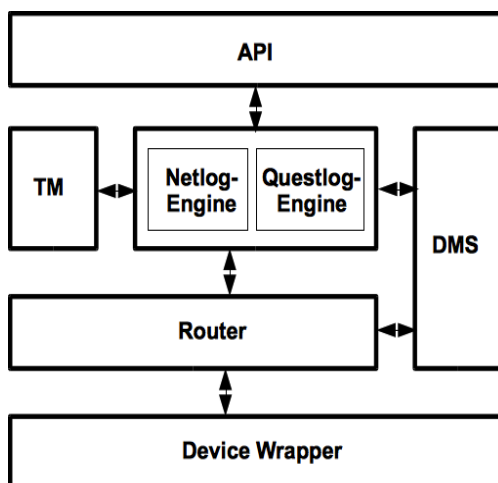


Figure 6.3: Netquest virtual machine architecture

We implemented an extended version of the *Netquest machine* [66, 29], Figure 6.3, which is initially used to execute *Netlog* programs following the *forward chaining* mechanism. The *Netquest virtual machine* is composed of six components that have been described in Chapter 4. In this section, however, we will explain in more details the innovations introduced to the Netquest system. Three important functionalities have been introduced (i) a *Router module* to evaluate intensional destinations and to communicate with the network, (ii) a *Questlog Engine* to execute the Questlog

6.3 System Architecture

queries and programs, following the *backward chaining* mechanism, and (iii) an *API* to allow firing queries at run time.

The Netquest virtual machine executes the bytecode, generated by the compiler, and manipulates data and messages. When a message is received by a device, the device wrapper transfers it to the router. The message is read by the router and the payload is sent to the corresponding engine if the device belongs to the destination. The engine loads appropriate rules and then evaluate these rules using the DMS. The DMS can update or delete data and create messages to be sent. These new messages are sent to the network through the device wrapper.

6.3.1 Router

As we have seen in Chapter 5, the Router is composed of two main modules: (i) Reception module that receives messages from the network, and (ii) Emission module that sends messages to other nodes in the network. Figure 6.4 shows the message format. The destination is composed of both *extensional* and *intensional* destinations, as seen in Chapter 3.

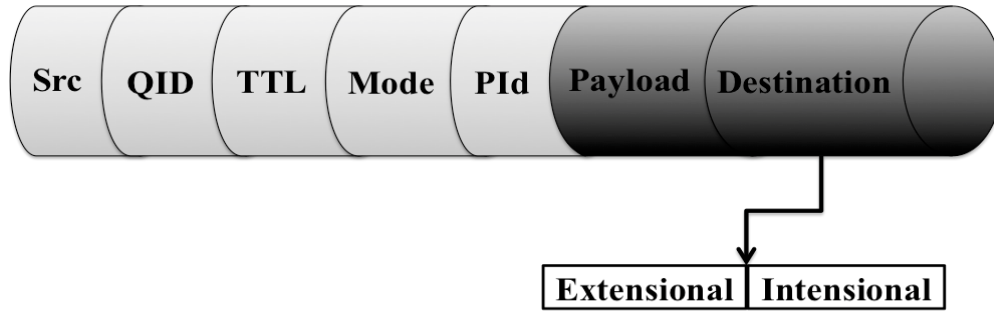


Figure 6.4: Message format

When a node receives a message, the router first checks if the node is the destination. Intuitively two cases have to be considered corresponding to extensional or intensional destination as shown in Algorithm 6.1.

- **Case 1:** The extensional destination is not empty. Then the Reception module checks if the node address is the extensional destination. In this case, the Reception module defines a unique identifier for the message and a unique identifier for the payload. Afterwards, the Reception module stores the message in reception bookKeeping and transfers the payload to the engine. Otherwise, the node address is not the destination, then the Reception module decreases the *TTL* of the message, and transfers it to the emission module of the router if the new *TTL* is greater than zero, and otherwise discards it.
- **Case 2:** The extensional destination is empty. Then the Reception module creates new message M_j that contains as payload the intensional destination of the initial message M_i , and as source the address of the router. Then the new message M_j is stored in the reception bookKeeping, and the payload of new message M_j is transferred to the Engine to be evaluated. When receiving the set of answer, the Reception module checks if the node address is in the set. If it is the case, the Reception module stores the initial message M_i in the reception bookKeeping, and transfers the payload of the initial message M_i to the engine. Otherwise the node address is not in the set, then the message is discarded. Received messages with intensional destinations are also transferred to the Emission module of the router to be sent to other nodes.

6.3 System Architecture

Data: Incoming message M_i from the network <pre> 1 if $M_i.extDest$ <i>is not empty</i> then 2 if $M_i.extDest$ <i>equals</i> $node.address$ then 3 Define a unique LocalId and PayloadId; 4 Store message M_i in reception bookKeeping; 5 Transfer $M_i.payload$ to the Engine; 6 else 7 Decrease TTL by 1; 8 if $TTL > 0$ then 9 Update TTL in M_i; 10 Transfer M_i to the Emission module; 11 end 12 end 13 else 14 if $M_i.intDest$ <i>is not empty</i> then 15 Create new message M_j set $M_j.payload = M_i.intDest$ and $M_j.source = router.address$; 16 Store message M_j in reception bookKeeping; 17 $\sigma_{Ans} = \text{Execute } M_j.payload \text{ through the engine}$; 18 if $node.address \in \sigma_{Ans}$ then 19 Store M_i in reception bookKeeping; 20 Transfer $M_i.payload$ to the Engine; 21 else 22 Discard message; 23 end 24 Transfer M_i to the Emission module; 25 else 26 Discard message; 27 end 28 end </pre>
--

Algorithm 6.1: Router Reception module

Let us move on to see the Emission module which is used to send messages to their appropriate destinations in the network. Intuitively two cases have to be considered according to extensional destinations of messages as shown in Algorithm 6.2.

- **Case 1:** The extensional destination is not empty. Either the extensional destination is *all*, then the message is transferred to neighbors, or the extensional destination is an address, then the message is send to the next hop on the route to the destination if the next hop if found and otherwise the message is discarded.
- **Case 2:** The extensional destination is empty. The message is transferred to neighbors.

6.3 System Architecture

Data: incoming message from the Reception module or from the Engine

```
1 if message.extDest is not empty then
2   if message.extDest is not all then
3     Fetch nextHop from routing table;
4     if nextHop is not empty then
5       Send message to nextHop;
6     else
7       Discard message;
8     end
9   else
10    Send message to neighbors;
11  end
12 else
13   Send message to neighbors;
14 end
```

Algorithm 6.2: Router Emission module

6.3.2 Questlog Engine

The Engine executes received queries with their corresponding answers based on related Questlog programs stored in the local data store. In addition, the Engine maintains data structures as shown in Section 6.1.4 to store mainly pending queries with their related queries identifiers, as well as the sources of queries.

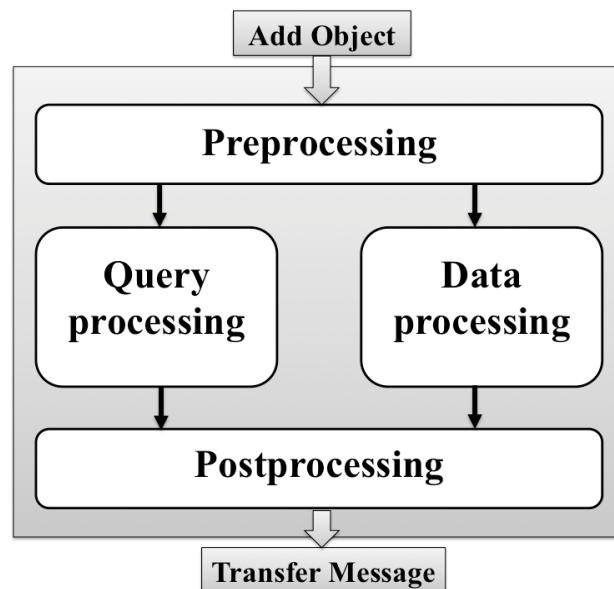


Figure 6.5: The Questlog Engine

Figure 6.5 shows the different modules of the the Questlog Engine. The input is an object that contains mainly a payload as well as other information such as the identifier of the related program that will be used to process the payload, an identifier of the payload, and a unique local identifier. The output is messages to be sent to their appropriate destinations. In the following, we present

6.3 System Architecture

the different modules of the Questlog Engine.

Preprocessing

This module analyses the incoming objects in particular the *payloads*, as shown in Algorithm 6.3. For each payload, 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.

```
input: An object that contains Payload, PId, PayloadId, LocalId  
1 Read and store the Payload in the payload bookKeeping, Table 6.11;  
2 switch Payload.content do  
3   | case Payload.content is a Query  
4   |   Transfer the Query to Query processing  
5   | case Payload.content is an Answer  
6   |   Transfer the Answer to Data processing  
7 endsw
```

Algorithm 6.3: Preprocessing module algorithm

Query processing

This module computes the queries, as shown in Algorithm 6.4. 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 SQL queries corresponding to matching rules 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.

```
input: A Query with PId, PayloadId, LocalId, and QId  
1 Store Query locally;  
2 Retrieve rules names by matching QueryName and PId with the attributes FactName and  
  PrId of the table Questlog pull rules, Table 6.4;  
  // SELECT PR.RuleName FROM QuestlogPullRules AS PR WHERE PR.PrId=PId AND  
  PR.FactName=QueryName;  
3 Load related SQL queries from QueryStore based on rules names;  
  // SELECT QS.Query, QS.FactName FROM QueryStore AS QS WHERE  
  QS.RuleName=RuleName AND QS.PrId=PId;  
4 Execute the SQL queries;  
5 Transfer results to module Postprocessing;
```

Algorithm 6.4: Query module algorithm

Data processing

This module handles received data as answers of queries, as shown in Algorithms 6.5 and 6.6. The SQL queries corresponding to matched rules are retrieved based on the rule name. The rule name is specified based on the identifier of the protocol, *PrId*, as well as the name of the relation of the received answer, *FactName*. The retrieved rules may contain forall ($\forall$). In this case, the local data structure, Table 6.14, is updated, and the SQL queries will not be executed till getting all answers.

6.3 System Architecture

Otherwise, the SQL queries are executed through the DMS and deduced facts are transferred to the module *postProcessing*.

```
input: PId, PayloadId, LocalId, QId, FactName, Src
1 Match FactName and PId with the attributes PrId and FactName of the table
  QuestlogPushRules, Table 6.5;
  // SELECT QPR.RuleName, QPR.ForAll FROM QuestlogPushRules AS QPR WHERE
    QPR.PrId=PId AND QPR.FactName=FactName
2 if forAll is empty then
3   | Call loadAndExecuteRules(RuleName, PId);
4 else
5   | // Update table ControlAnswerForAll, Table 6.14
    UPDATE ControlAnswerForAll SET AnswerLocalId=LocalId WHERE QueryId = QId
    AND Destination=Src;
    // Check if all answers are received
6   SELECT AnswerLocalId, Destination FROM ControlAnswerForAll WHERE
    QueryId=QId;
7   foreach AnswerLocalId do
8     | if AnswerLocalId is null then
9       | if Neighbor always exists in neighborhood table then
10      | | Don't compute the fact;
11      | | exit;
12      | end
13    | end
14  end
15  Call loadAndExecuteRules(RuleName, PId);
16  DELETE FROM ControlAnswerForAll WHERE QueryId=QId;
17 end
```

Algorithm 6.5: Data module algorithm

```
input: RuleName, PId
1 Load related SQL queries from QueryStore using RuleName;
  // SELECT QS.Query, QS.FactName FROM QueryStore AS QS WHERE
    QS.RuleName=RuleName AND QS.ProtocolId=PId
2 Execute the SQL queries;
3 Transfer results to module Postprocessing;
```

Algorithm 6.6: Load and execute rules algorithm

Postprocessing

This module generates a *payload* in Questlog form by collecting subqueries or facts, fetches their corresponding destinations, encapsulates them in messages, and finally transfers the messages to

6.3 System Architecture

the Emission module of the router. Algorithm 6.7 shows the algorithm of *Postprocessing* module.

Data: Results of query/answer computation

```

1 while Queue is not empty do
2   Get first element from the Queue;
3   Create a payload set Payload.content = related Result;
4   Create and encapsulate a message;
5   Transfer the message to the Emission module of the router;
6 end

```

Algorithm 6.7: Postprocessing module algorithm

6.3.3 Application Programming Interface and Code Editor

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. Bellemon *et al.* [29] proposed the QuestMonitor visualization tool, Figure 6.6, which allows to interact with a network on a 2D graphical interface and visualizes the behavior of declarative protocols.

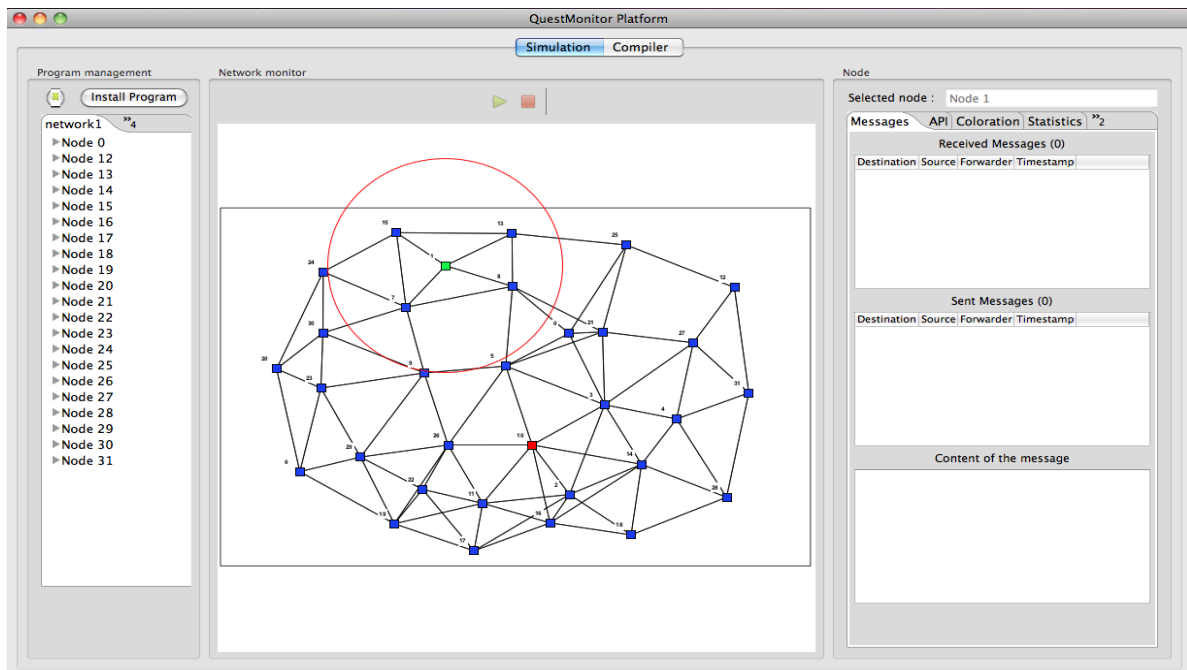


Figure 6.6: Overview of the QuestMonitor user interface

The QuestMonitor is initially used with the Netlog language. We have modified the API of the QuestMonitor in order to allow selected node to send Questlog queries in the network at run time. Figure 6.7 shows the API where we select a node that sends the query (e.g. *Node 1*), 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)*).

6.3 System Architecture

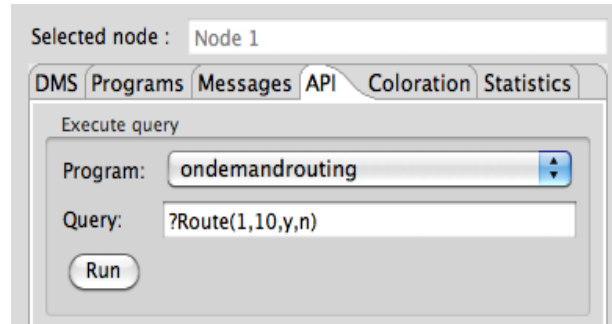


Figure 6.7: Application programming interface

When running a query from the API, it is transferred to the Questlog Engine of the node where execution is taking place to be evaluated. The Questlog Engine then computes the related query. For instance, upon running the query $?Route(1, 10, y, n)$ from the API as seen in Figure 6.7, it is transferred to the Questlog Engine of node source "Node 1" to be evaluated.

To facilitate the programming of Questlog programs and to ensure their compilation, we have developed a code editor, as seen in Figure 6.8, which is an environment for helping developers to write programs. The code editor offers standard functionalities such as syntax coloring and error detection.

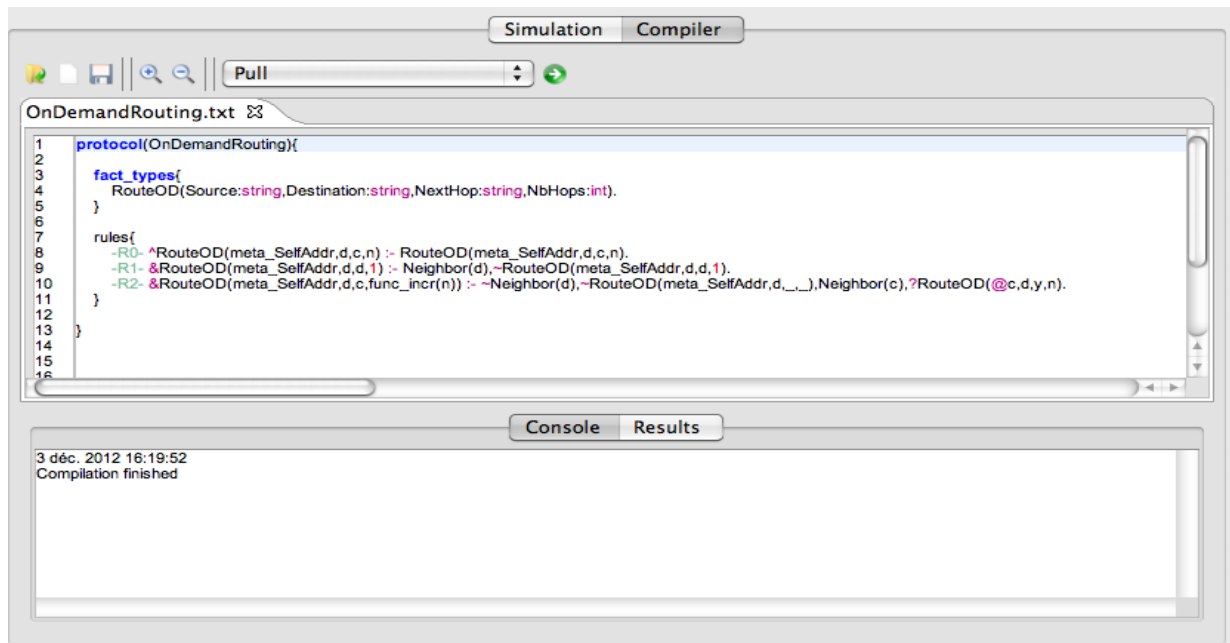


Figure 6.8: Code editor

After compilation of a program, a set of SQL queries is obtained as seen in Figure 6.9. The result is saved in a file that should be installed on each node of a network before running the program. It is worth noting to mention that different programs can be run on the same node. In this case, the programs should be compiled and installed on nodes.

Protocols with Intensional Destinations

Contents

Introduction	109
7.1 Motivation	110
7.2 Data Collection using Questlog	114
7.2.1 Sensor Data Collection	114
7.2.2 One-hop Data Aggregation	114
7.3 Cluster-based Data Aggregation	115
7.3.1 Dynamic Intensional Clustering	116
7.3.2 Aggregated Data Transfer	121
7.4 Experiments over QuestMonitor	124
7.4.1 Load Balancing	124
7.4.2 Dynamic Adaptation	126
7.4.3 Characteristics of Clusters	127
7.4.4 Particular Case	127
7.5 Discussion	128
Conclusion	129

Introduction

In Chapter 3, we presented a framework which allows messages with intensional destinations. We demonstrated in Chapter 4 the framework in a special class of peer-to-peer systems using the first method of execution based on extensional destination higher priority order. In this chapter, we demonstrate the framework in the domain of wireless sensor networks (WSNs) using the second method of execution based on intensional destination higher priority order.

Recent advances in miniaturization and low-power design have led to the development of small-sized battery-operated sensors that are capable of monitoring and detecting a variety of ambient conditions such as temperature, pressure, humidity, movement, noise, and lighting. Sensors are generally equipped with data processing and communication capabilities. The sensing circuitry measures parameters from the environment surrounding the sensor and transforms them into an electric signal [2]. Each sensor has an onboard radio that can be used to send the collected data to interested parties (e.g. base station, sink). Sensors present constraints on power consumption, as well as on processing, memory, and wireless communication capabilities.

In wireless sensor networks, thousands of small sensor nodes can be quickly deployed in a vast field. Sensor nodes collect data and then transmit streams of data to a common base station or sink node. Data transmission can take place either in the push mode, where sensors actively send their data to sinks, or in the pull mode, where sensors transmit their data only upon sinks' requests. For instance, a sink node may fire a query such as "Is the temperature greater than T "? A subset of sensor nodes may satisfy the query. Then such kind of queries are disseminated in messages based on their content. Our framework is well suited for applications that allow such queries with destination specified intensionally. Declarative queries are especially relevant for sensor network interaction: Users and application programs issue queries without knowing how the data is generated in the sensor network and how the data is processed to compute the query answer. Declarative queries can be expressed in the Questlog language. We show in Section 7.2.1 the use of the language Questlog to collect data from a network.

Since sensor nodes are usually powered by batteries, increasing network lifetime is a major goal of any sensor network system. Data transmission to a central node for offline storage, querying, and data analysis is very expensive for sensor networks since communication using wireless medium consumes a lot of energy [131]. Since sensor nodes have local computation abilities, part of the computation can be moved into the sensor network, aggregating or eliminating irrelevant records. In-network processing can reduce energy consumption and improve sensor network lifetime significantly compared to traditional centralized data extraction and analysis [163]. In Section 7.2.2, we use Questlog to express queries that allow to aggregate efficiently data in a network.

To reduce the delay, to balance the load, and to minimize the traffic cost, some approaches [158, 146, 114, 70, 121] propose to use clustering. Heinzelman *et al.* introduces LEACH [70], a protocol in which each node decides to be a cluster head with a probability p , and broadcasts its decision. Each non cluster head node chooses the cluster head that can be reached using the least communication energy. The role of being a cluster head is rotated among the nodes to balance the load. In [166], Zhang and Arora present GS³, an algorithm for self-configuring a wireless network into a cellular hexagon structure, formed by dividing the area into cells of equal radius. One special node starts the clustering process by selecting the heads of neighboring cells. Unselected nodes become cell members. This process is repeated by each cell head. GS³ is self-healing, and it is applicable for static and dynamic networks. In [146], Soro *et al.* presents an approach for cluster-based network organization based on a set of coverage-aware cost metrics that favor nodes deployed in densely populated network areas. The idea is that nodes in sparsely deployed areas, as well as nodes with

7.1 Motivation

small remaining energies are used less often as data routers, so that these nodes can collect data for longer periods of time. Wang *et al.* [158] investigates the maximization of the amount of data gathered during the lifetime of a cluster. The authors present a method to find the near optimal transmit power of each cluster member, and introduce an algorithm which determines the optimal cluster head that provides the largest amount of collected data. Nikaein *et al.* proposes DDR [121] to make clusters without cluster-head. The formation of clusters is based on the construction of a tree. Each node selects as parent its neighbor with the highest connectivity (degree). All nodes in the same tree belong to the same cluster. The algorithm of DDR was then taken by Baccelli [21] by adding the notion of cluster-head and by controlling the size (e.g. d hops) of the clusters by disseminating information along the branches of the tree. If a branch is too long, the node at $d + 1$ hops should select another parent. In [114], Mitton *et al.* introduces a measure (density) that allows to form clusters and perform the cluster head election. The density criteria reveals to be stable when the topology slightly evolves. The idea is to recompute the clusters topology as less as possible in spite of nodes mobility. But as we will see in Section 7.1, creating clusters, electing and maintaining cluster heads, notifying periodically all nodes in the same cluster, and balancing the load in intra-cluster are complex tasks and have non-trivial communication costs. In Section 7.3, we propose a clustering protocol based on intensional addresses, in which the network is decomposed into dynamic clusters where cluster heads are virtual nodes, not known *a priori* by any node in the network. The cluster head is a node that satisfies certain selection criteria specified intensionally. The cluster head is selected on the fly by evaluating the intensional destination, based on local data of 1-hop neighbors maintained on each node in the network.

The Chapter is structured as follows. In the next section, we present the different methods used to collect data and motivate the use of intensional destinations. In Section 7.2, we present programs expressed in Questlog using intensional destinations to collect data from a network. We describe in Section 7.3 our virtual clustering protocol, while we show in Section 7.4 some experiments over QuestMonitor. Section 7.5 is devoted to discussion and analysis of the proposed protocol.

7.1 Motivation

Wireless sensor networks (WSN) are autonomous and self-organizing systems consisting of a large number of sensor nodes and a limited number of sinks. Sensor nodes have strong restrictions in terms of energy since each sensor uses a battery typically not rechargeable, while sinks have constant power supply. Sensors are deployed randomly in the area of interest. A wide range of real-world applications [155], including: habitat and structural monitoring, surveillance, disaster management, inventory management, target tracking, has been deployed nowadays.

Sensor nodes gather useful information related to the surrounding environment (e.g. temperature, humidity, seismic and acoustic data), and transmit their sensed data to a base station (sink) for further processing. Various methods are used to let the sensors know the address of a sink. For instance, (i) it is stored as metadata in each sensor, (ii) the sink broadcasts an advertisement packet holding its identifier, (ii) the sink broadcasts a message to build a cost field [165], or similarly (iv) to build hierarchical levels [79], etc.

Data transfer: Using the conventional flat topology, each sensor node reports sensed data (either periodically or upon event-detection) and forwards them to the sink. Data transmission can be achieved using a direct communication protocol where each sensor node transmits its sensed data directly to the sink. This transmission is not always realistic assumption [2] since the sink is often not directly reachable to all sensor nodes due to signal propagation problems (e.g. the presence of

7.1 Motivation

obstacles), and requires a large amount of transmission power to communicate the data, especially if the appropriate sink is far, resulting in high energy dissipation and reduction of the network lifetime. Another approach can be used to transmit the data to the sink based on multi-hop routing protocols, where sensed data can be transmitted through intermediate nodes that act as sensors as well as relays. Using routing protocols to build and maintain routes to explicit destinations (e.g. sinks) are costly. In addition to battery sensor constraint, faults in such networks occur frequently, typically because sensor nodes are prone to failures due to natural phenomena such as rain, fire, as well as to adversary environmental manipulation. The high density of sensor networks leads to an increase of the number of transmitted messages to the sink, yielding a rapid depletion of sensor batteries and consequently to a reduced network lifetime.

Data aggregation and clustering: Since adjacent sensors often detect common phenomena, there might be some redundancy in the data communicated to the sink. In-network filtering and processing techniques such as data aggregation can help to conserve the scarce energy resources. The idea is to combine the data coming from different sensors to eliminate redundancy, minimize the number of transmissions and thus save energy. In order to aggregate data, achieve high energy efficiency and increase the network scalability, sensor nodes can be organized into clusters [69, 158, 2, 120, 21, 114, 57]. Every cluster has a leader, often referred to as the cluster head. One of the nodes in a cluster is selected as the cluster head, and the remaining nodes are cluster members. The cluster heads are of the same entity as the deployed sensors. The cluster head is usually in charge of certain local coordinations, such as collecting data from cluster members, aggregating the data, and communicating aggregated results to the sink. The cluster members transmit their sensed data to the cluster head. The challenges of clustering is to: (i) efficiently select the cluster head while minimizing the overhead messages and the energy consumption, (ii) balance the charge to maximize the network lifetime, and (iii) form and maintain a disjointed group of nodes in a distributed manner.

In classical clustering approaches [69, 158, 42, 81, 24, 2], the network is organized into a set of clusters. Sensor nodes exchange messages in order to elect cluster heads, and to form and maintain clusters. The cluster heads in some approaches notify their members so that each member knows the identifier of the cluster head. In dynamic topology, nodes may join or leave the network as well as they may fail. In case of failures, either the election process is triggered to elect new cluster heads and form new clusters, or a special mechanism (e.g. self-healing) is used [166] to maintain the clusters. However, these solutions have high cost due to increased number of control messages and backup copy. After cluster construction, the members transfer their data to their corresponding cluster heads. Each cluster might have a particular structure such as a tree [121, 120, 21, 114, 57]. Then the network can be seen as a set of trees where the cluster heads are the roots of the trees.

Scenario of a classical algorithm: Consider a cluster $\mathcal{C}$ which contains a set of cluster nodes $\{\alpha, \beta, \gamma, \dots\}$ and a cluster head $\mathcal{CH}$. Suppose member α has data to send, it encapsulates its sensed data in a message m , sets m 's destination to $\mathcal{CH}$ known *a priori*, and sends the message m to the cluster head $\mathcal{CH}$. The message travels in multi-hop routing (e.g. cluster $\geq k$ -hop and $k \geq 2$; a member is at maximum k hops away from the cluster head) until the cluster head. Suppose intermediate node β has no route to the cluster head $\mathcal{CH}$, in this case node β returns back the message m to the source node α , and maintains the cluster. The source node upon getting the message m , finds a new route to the (potentially new) cluster head, and then sends to it the message. Algorithm 7.1 shows the pseudocode of this scenario.

7.1 Motivation

```

input : Message to send to the cluster head
1 if Message.Destination = self then
2   | Store data in a local data structure;
3 else
4   | nexthop  $\leftarrow$  Fetch nexthop on the route to Message.Destination;
5   | if nexthop is not empty then
6   |   | Send Message to nexthop;
7   | else
8   |   | Send back the Message to the source node;
9   |   | Maintain the cluster;
10  | end
11 end

```

Algorithm 7.1: Pseudocode of a classical algorithm

In a static or pseudo-dynamic topology, the clusters are more likely to be stable as well as their appropriate cluster heads. This scenario is well suited for such topology since the election process of the cluster heads is done only once. However, this scenario has no or low load balancing. In addition, this scenario considers a priori knowledge of the cluster head by each member in the corresponding cluster. That results in a high complexity due to the increased number of messages needed to construct and maintain the clusters and accordingly the cluster heads. In dynamic topologies, this scenario leads to an increased cost due to the dynamcity of nodes which might be out of order at any time. There is thus a need of an energy-efficient algorithm that can adapt dynamically under topology changes.

Scenario of an adaptive algorithm using static code: We next present at a high level a scenario of an adaptive algorithm that allows to construct on the fly dynamic clusters when the messages travel. We consider that the network is decomposed into a set of clusters that are dynamically constructed and organized as a set of trees. The cluster heads are not known *a priori*. In this scenario, as shown in Algorithm 7.2, each node, say α , based on local data, sends its sensed data in a message to a neighbor node (its parent) which satisfies certain selection criteria (e.g. high energy, high degree, lowest Id). If node α selects node β , we say that node β is the parent of node α , noted $\mathcal{P}(\alpha) = \beta$. The parent repeats the same process until the cluster head, say η . The cluster head is the node that has the highest selection criteria between its neighbors, $\mathcal{P}(\eta) = \eta = \mathcal{CH}$. The cluster head is the root of the tree. As a result, we obtain a set of trees that form non-overlapping clusters. The cluster head stores received data in a local data structure.

In contrast to classical clustering approaches, Algorithm 7.2 has no set up phase to construct the clusters. Instead, the clusters are constructed on the fly. In addition, this algorithm does not require a mechanism such as self-healing to maintain the clusters, it adapts dynamically to topology changes. When a message is received or a sensed data is to be sent, a node evaluates locally Lines (1-7) of Algorithm 7.2 to obtain an identifier of a node to which the message will be sent. In classical approaches, a node needs to fetch the nexthop to route the message to the cluster head. In our approach, however, each node evaluates locally the code, which might result in additional computation complexity. It is noteworthy to mention that Yao *et al.* mentioned that communication using the wireless medium consumes a lot of energy, while local computation is cheap [163].

7.1 Motivation

```

input : Message to send to the cluster head
1 Set  $\mathcal{CH} = \text{self}$  ;
2 Search for a  $\mathcal{CH}$  (parent) ;
3 foreach neighbor  $n_i$  do
    // Compare selection criteria (SC) and choose a cluster head
4   if  $SC_{n_i} > SC_{\mathcal{CH}}$  then
5      $\mathcal{CH} = n_i$ ;
6   end
7 end
8 if  $\mathcal{CH} = \text{self}$  then
9   | Store data in a local data structure;
10 else
11   | Update Message set  $\text{Message.Destination} = \mathcal{CH}$ ;
12   | Send Message to  $\mathcal{CH}$ ;
13 end

```

Algorithm 7.2: Pseudocode of an adaptive algorithm

Scenario of an adaptive algorithm using mobile code: In Algorithm 7.2, we have seen an adaptive algorithm to construct seamlessly a set of clusters that allow to aggregate data efficiently. However the code of the algorithm should be stored and installed at each node in the network. The code is planned and designed *a priori*. It is then difficult to change or modify the code. In his seminal paper [156], Wall said that "*the difference between a clear algorithm and an obscure one is often no more than a matter of finding the right viewpoint from which to describe it.*" The right viewpoint might lead to a simple, modular, and elegant formulation. We propose to use active messages that hold the code of the selection criteria. That means the code that allows to select the cluster head or the parent of a node is mobile. Our framework shown in Chapter 3 offers a new model of messages whose destination specified both extensionally and intensionally. The intensional destination is represented by a selection criteria. Thus in this scenario, when the message is traveling in the network, the intentional destination is evaluated on the fly at each node upon receiving the message, as shown in Algorithm 7.3. The mobility of the code allows to change dynamically the selection criteria, and facilitates the programming of applications. This formulation is slightly simpler, Algorithm 7.3, in the sense that it is shorter and not as deeply nested. But more importantly it makes it adaptive and clearer that the goal is to get the message to the cluster head.

```

input : Sensed data or a Message to transfer to the cluster head
1  $\mathcal{CH} \leftarrow$  Execute the intensional destination;
2 if  $\mathcal{CH} = \text{self}$  then
3   | Store data in a local data structure;
4 else
5   | Update Message set  $\text{Message.ExtensionalDestination} = \mathcal{CH}$ ;
6   | Send message to  $\mathcal{CH}$ ;
7 end

```

Algorithm 7.3: Pseudocode of an adaptive scenario using (mobile) intensional destination

The idea of programming using active messages has been proposed long ago in [156], where network programs are encapsulated in active messages traveling in the network. It provides a simple and elegant way to describe and understand distributed programs.

7.2 Data Collection using Questlog

In this section, we present programs expressed in the Questlog language to collect and aggregate data from a network. We start by a simple program that allows to collect data from all sensors, Section 7.2.1. We then present a program that allows only a subset of nodes to aggregate data from their one-hop neighbors before sending their aggregating results to the sink, Section 7.2.2.

7.2.1 Sensor Data Collection

Consider an application where a sink node S floods a query $?GetData(x, t)$ to all sensors to collect on-demand the temperature value t of sensor nodes. Upon receiving the query, each sensor node forwards its sensed data to the sink. This query can be mapped to a rule-based program which models its semantic. This program consists on only a unique trivial rule, Rule (7.1).

$$\uparrow GetData(x, t) : - Tmp(x, t). \quad (7.1)$$

Schema	Description
GetData(x,t)	GetData(sensorId, temperature)
Tmp(x,t)	Tmp(sensorId, temperature)

Table 7.1: Schemas of the *sensor data collection* program

Each node, say ν , stores its temperature in the relation Tmp . Upon receiving the query, the Questlog Engine of node ν uses Rule (7.1) to evaluate it. Deduced result $GetData(\nu, t)$ which includes the temperature of the sensor is sent ($\uparrow$) in unicast mode following the reverse path towards the sink S .

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

7.2.2 One-hop Data Aggregation

We have seen in Section 3.1 that a message may contain a *content-query* and a *dest-query*. In this section, we explain the use of destination queries. Assume the sink node S sends a message that contains (i) a *content-query* in the payload, and (ii) a *dest-query* in the destination. We have seen in the previous example, with Rule (7.1), that data collection might involve all nodes in a network. However, due to sensor power constraints, it might be preferable [85] that data collection be performed only from a subset of nodes.

Assume that the sink node S calls sensor nodes that have energy level ℓ greater than a threshold η to be considered as cluster heads to collect data (e.g. temperature) from their one-hop neighbors, aggregate the received data, and then send the aggregated value to the sink. The sink S sends a message with *content-query* $?Collect(x, t)$ and *dest-query* $?Powerful(x)$ in the network. Suppose that the energy level is stored in the relation *Energy*. The following program, Rules (7.2 - 7.4),

7.3 Cluster-based Data Aggregation

defines its semantics:

$$Powerful(x) : - Energy(\ell), \ell > \eta. \quad (7.2)$$

$$\uparrow Collect(x, avg(t)) : - \forall y Link(x, y), ?GetData(@y, t). \quad (7.3)$$

$$\uparrow GetData(x, h) : - Tmp(x, h). \quad (7.4)$$

Schema	Description
Powerful(x)	Powerful(sensorId)
Energy(ℓ)	Energy(level)
Collect(x,t)	Collect(sensorId, temperature)
Link(x,w)	Link(source, destination)

Table 7.2: Schemas of the *one-hop data aggregation* program

Each sensor node, say α , upon receiving the message evaluates first the *dest-query* $?Powerful(\alpha)$ using Rule (7.2) after matching the head of the rule. If the body of the rule $Energy(\ell), \ell > \eta$ is satisfied, then the sensor node α belongs to the destination, and is now allowed to evaluate the *content-query* $?Collect(\alpha, t)$. Otherwise, sensor node α discards the message. It is noteworthy to mention that this model intuitively allows to save energy and maximizes the lifetime of the related network because only sensor nodes that satisfy the *dest-query* are enabled to evaluate the *content-query*, which is more complex and requires communication.

The *content-query* $?Collect(\alpha, t)$ matches the head of Rule (7.3), which leads to the evaluation of its body $\forall y Link(x, y), ?GetData(@y, t)$ that gives raise to queries $?GetData(@y, t)$ sent to all neighbors y . Each neighbor upon receiving the query $?GetData(y, t)$, uses Rule (7.4) and returns its temperature value to the source α . When all answers ($\forall$) are received, node α resumes the evaluation of Rule (7.3) in the push mode, leading to a fact $Collect(\alpha, t)$ where t in this case is the average temperature, to be sent ($\uparrow$) in unicast mode following the reverse path towards the sink S .

7.3 Cluster-based Data Aggregation

We have seen in Section 7.1 that classical clustering approaches require an election process to choose cluster heads and form clusters, and in some approaches periodic notification from cluster heads to all cluster members. In dynamic topology, the election process is repeated periodically, or triggered upon failure detection. The dynamicity might lead to loss of messages traveling in the network either to the cluster head or to the sink. Maintaining the clusters and minimizing the loss of data in messages using special mechanism such as self-healing are thus costly in terms of communication complexity, and result in an increase of the number of exchanged messages. The main objectives of clustering are to balance the load, provide fault-tolerance, reduce the delay, reduce the traffic cost, and thus maximize the network longevity. One of the major problems in classical approaches is the passive knowledge of the cluster heads. Each time a sensor node that needs to send its collected data, should know *a priori* the identifier of its cluster head. It would be interesting if we abstract the cluster heads. Each sensor node can send its collected data intensionally to the sensor node (cluster head) that satisfies a specified selection criteria. This way can facilitate the tasks and satisfy as much as possible the objectives of clustering.

In this section, we present a Dynamic Intensional ClustEring (DICE) protocol that allows, based

7.3 Cluster-based Data Aggregation

on intensional addressing, to collect and aggregate data in an simple and modular manner. The basic idea of the DICE protocol is to aggregate efficiently collected sensors data, by constructing on the fly, based on intensional destinations, a set of dynamic trees from a network topology. Each sensor node chooses on the fly its parent by evaluating certain selection criteria when the messages are traveling. A node calculates on the fly the identifier of its parent based on its local data of 1-hop neighbors maintained regularly. The parent is the sensor node which has the highest selection criteria. The root is the sensor node that obtains itself as a parent, $\mathcal{P}(\mathcal{H}) = \mathcal{H}$. Each tree forms a cluster. The root of each tree is the cluster head. Then, the network is virtually partitioned into a set of non-overlapping dynamic clusters. The selection criteria is based on the degree (number of neighbors) of a node. In case of ties, the sensor node with the lowest identifier is chosen. It is noteworthy to mention that any other criteria can be used, as well as combinations of different criteria. Then the size of a cluster increases and decreases dynamically depending on some network features such as node density, rate of network connection/disconnection, and transmission power.

Collected data by the cluster heads are aggregated and transferred upon sink's requests. The sink periodically sends requests with intensional destinations, asking only the cluster heads to aggregate their collected data and send aggregated results to the sink.

7.3.1 Dynamic Intensional Clustering

In this section, we give the preliminary notations to describe the proposed protocol DICE [14]. Then we present the different phases of the dynamic construction of the clusters.

Preliminaries and notations

A wireless multi-hop network is modeled by a graph $G = (V, E)$, where V being the set of nodes and $E \subseteq V^2$ is the set of communication edges. An edge exists if the distance between two nodes u and v is less or equal than a fixed radius R , which represents the radio transmission range:

$$E = \{(u, v) \in V^2 \mid u \neq v \wedge uv \leq R\}, \quad (7.5)$$

uv being the Euclidean distance between u and v . Accordingly, the neighborhood of node u is defined by the set of nodes that are inside a circle with center at u and radius R , and it is denoted by:

$$\Gamma_R(u) = \Gamma_u = \{v \mid (u, v) \in E\} \quad (7.6)$$

Notice that node u does not belong to its neighborhood ($u \notin \Gamma_u$). The degree of a node u in G is the number of edges which are connected to u , and it is equal to:

$$\Delta(u) = |\Gamma_R(u)| \quad (7.7)$$

Exchange of *hello* message

We assume that each node u generates periodically a *hello* message to the neighboring nodes that are within its direct radio transmission range. Initially, the *hello* messages contain the identifier of the source node. When receiving a *hello* message, a node inserts the identifier of its neighbor, say α , in a table *Neighbor* with a timeout t usually equal to two periods, if α does not exist. Otherwise, it updates the entry corresponding to α . Periodically, an entry is deleted if its timeout is expired.

7.3 Cluster-based Data Aggregation

As we have seen in Chapter 3, the destination of messages is composed of both extensional and intensional destinations. The intensional destination is specified by a selection criteria which in this chapter is based on the degree of nodes. We extend the neighborhood table, Table 7.3, with additional attribute corresponding to the degree. A node exchange thus *hello* message with two fields: $\langle NeighborId, Degree \rangle$, where *NeighborId* is the identifier of node u , and *Degree* is its degree $\Delta(u)$. The time intervals between two *hello* messages should depend on some network features like rate of network connections/disconnections.

Table Neighbor		
NeighborId	Degree	Timeout
int	int	int

Table 7.3: Neighborhood table

Cluster head selection

Recall that the construction of the trees (clusters) is on the fly. When a node, say u , has data to send, it chooses a parent (can be seen as local cluster head to the node) based on its local data. The parent is the node that satisfies the specified selection criteria (degree). The node with the maximum degree is chosen as parent. In the case of ties, the node with the smallest identifier is chosen. Indeed, node u searches in its table *Neighbor* a set of nodes as parent candidates (PC) whose degrees are equal to maximum neighborhood degree, and greater or equal than to u 's degree. This set is denoted by:

$$PC_u = \{v \mid \Delta(v) = \max(\Delta(\Gamma_v)) \wedge \Delta(v) \geq \Delta(u)\}$$

We distinguish between three cases:

1. If the set PC_u is empty, then two cases have to be considered:
 - Node u has no neighbors, and thus no parent candidate. In this case, it is an isolated node which considers itself as parent as well as the cluster head. In Figure 7.1, node 1 has no neighbor, then $\mathcal{P}(1) = 1 = \mathcal{CH}$.
 - Node u has a set of neighbors but the degree of node u is the highest. In this case, node u is the root of the tree, and consequently it is the cluster head. In Figure 7.1, node 2 has four neighbors: $\{10, 5, 9, 6\}$ but the set of PC_2 is empty, then node 2 is the cluster head, $\mathcal{P}(2) = 2 = \mathcal{CH}$.
2. If the set PC_u has only one member w , then two cases have to be considered:
 - if $\Delta(w) > \Delta(u)$, then this member w is the elected parent. For example, in Figure 7.1, node 14 has three neighbors: $\{7, 4, 12\}$ but the set of PC_{14} has only one member which is the node 4. Since $\Delta(4) > \Delta(14)$, then $\mathcal{P}(14) = 4$.
 - if $\Delta(w) = \Delta(u)$, then the node with the lowest identifier is the elected parent. For example, the set of PC_{15} has only one member which is the node 4. Since $\Delta(4) = \Delta(15)$, then $\mathcal{P}(15) = 4$.
3. If the set PC_u has more than one member, this means that there are more than one neighbor, say x , with the maximum degree. Then two cases have to be considered:

7.3 Cluster-based Data Aggregation

- if $\Delta(x) > \Delta(u)$, then node u elects as parent the node with the lowest identifier from the set PC_u . In Figure 7.1, node 5 has two neighbors: $\{15, 2\}$ with the same degree. Thus it elects as parent node 2 which has the lowest identifier, $\mathcal{P}(5) = \min(PC_5) = 2$.
- if $\Delta(x) = \Delta(u)$, then node u elects as parent the node with the lowest identifier, which might be u 's identifier.

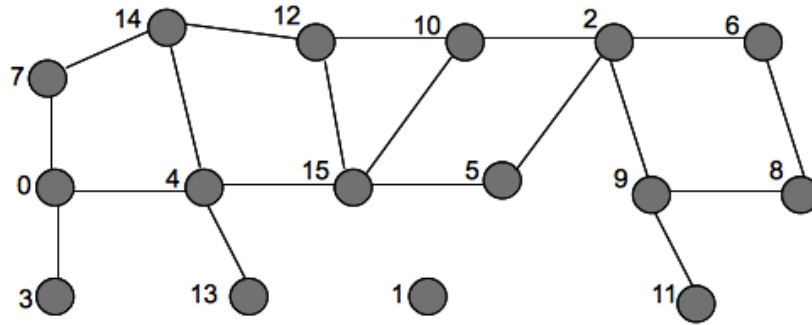


Figure 7.1: A topology example

Each node chooses intensionally its parent when it has data to send. As a result, a set of trees is obtained. Each tree forms a cluster, as shown in Figure 7.2.

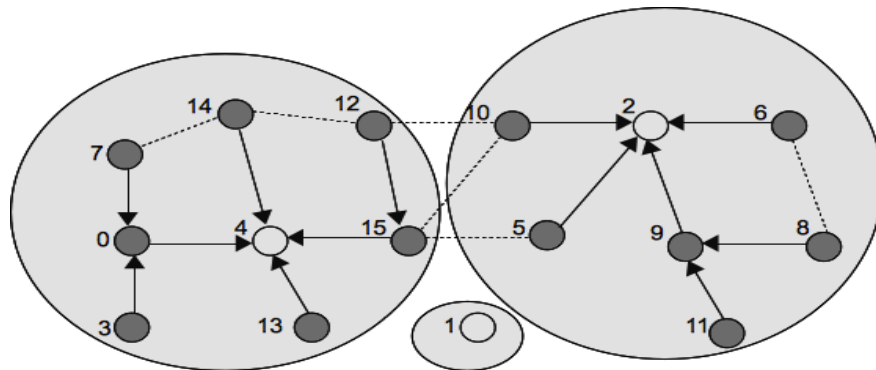


Figure 7.2: Clustering of trees where roots (cluster heads) are shown as white nodes and branches are shown as arrows

Operational semantics of dynamic tree clustering

In addition to their sensing capabilities, the architecture of sensor nodes, as shown in Figure 7.3, is composed of three main modules: (i) an Engine, to execute applications and routing programs, (ii) a Data store, to save all data related to applications and routing, and (iii) a Router, to communicate with other sensor nodes in the network. As we have seen in Chapter 5, the Router is composed of two (sub)modules: (i) Reception module, to handle received messages from the network and (ii) Emission module, to send messages in the network.

Before starting the description of the different modules, we should note that in this section, we treat only the messages that contain sensed data (also known as packet of type data). Other type

7.3 Cluster-based Data Aggregation

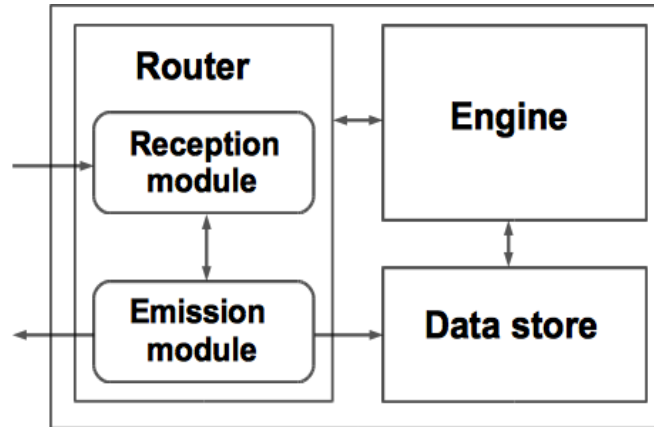


Figure 7.3: A sensor node architecture

of messages (e.g. aggregation) will be treated in Section 7.3.2. As shown in Algorithm 7.4, when a sensor node α has (sensed) data to send, the *Engine* of node α creates a message, sets up the message's payload with the data, and the message's intensional destination with the code of the cluster head selection criteria, say CHSC. The extensional destination of the message is still empty since in our approach there is no prior knowledge of the identifier of the cluster head. Node α then transfers the message to the *Emission module* of the router, Line (5) in Algorithm 7.4.

Let us now describe the behavior of the *Engine* of a cluster head node upon receiving a *payload* from the *Router*. It simply stores temporary the data in a table called *Aggregation*, as shown in Lines (6 – 7) in Algorithm 7.4.

```

input: sensed data
1 switch Node.type do
2   case Node.type as a member
3     Collect sensed data;
4     Create Message set Message.Payload = sensed data, Message.type = data and
      Message.intDest = CHSC;
5     Transfer Message to the Emission module;
6   case Node.type as cluster head
7     Store sensed data in table Aggregation;
8 endsw

```

Algorithm 7.4: Functionalities of the Engine of a sensor node

We now describe the behavior of the *Emission module* of node α when it receives the message (with empty extensional destination) from the Engine. It transfers the message to the *Reception module*, as shown in Line (13) in Algorithm 7.5. Otherwise, the extensional destination is not empty, then it routes classically the message by fetching and sending the message to the parent (nexthop), Lines (5 – 7). In dynamic network, the parent may leave the network, in this case we recompute the intensional destination to get a new parent. This is done by setting up the message's extensional destination to empty, and transferring it to the *Reception module*, as shown in Lines (9 – 10) in Algorithm 7.5.

7.3 Cluster-based Data Aggregation

```

input: Message
1 foreach Message do
2   switch Message do
3     case Message.type is sensed data (packet of type data)
4       if Message.extDest is not empty then
5         Get nextHop to the destination extDest from routing table;
6         if nextHop is not empty then
7           Send Message to nextHop;
8         else
9           Update Message set Message.extDest to empty;
10          Transfer Message to the Reception module;
11        end
12      else
13        Transfer Message to the Reception module;
14      end
15    endsw
16 end

```

Algorithm 7.5: Router Emission module: : packet of type data

Let us now describe the behavior of the *Reception module* of node α . For each message, it first executes the intensional destination, as shown in Line (3) in Algorithm 7.6. If the computation *Result* is empty, then node α is the cluster head, $\mathcal{P}(\alpha) = \alpha = \mathcal{CH}$. Thus, the *Reception module* transfers the content of the message to the *Engine*, Line (5) in Algorithm 7.6. Otherwise, the computation *Result* contains an identifier of a node, say β , corresponding to the parent of α , $\mathcal{P}(\alpha) = \beta$. Note that a relation *Parent* is installed on each node. This relation contains a unique entry corresponding to the parent of the node. Thus, the *Reception module* updates the parent relation of node α , noted $Parent(\alpha) = \beta$. At the same time, it updates the message's extensional destination to β , and transfers the message to the *Emission module*.

```

input: Message
1 foreach Message do
2   case Message.type is sensed data (packet of type data)
3     Result  $\leftarrow$  Execute intDest;
4     if Result is empty then
5       // node  $\alpha$  is the cluster head
6       Transfer Message.payload to the Engine;
7     else
8       Update parent set  $Parent(\alpha) = Result$ ;
9       Update Message set Message.extDest = Result;
10      Transfer Message to the Emission module;
11    end
12  end
13 end

```

Algorithm 7.6: Router Reception module: : packet of type data

In Figure 7.4, we show a small network where we run the dynamic clustering protocol DICE. The network is dynamically organized into a set of trees as clusters when sensed data are traveling into

7.3 Cluster-based Data Aggregation

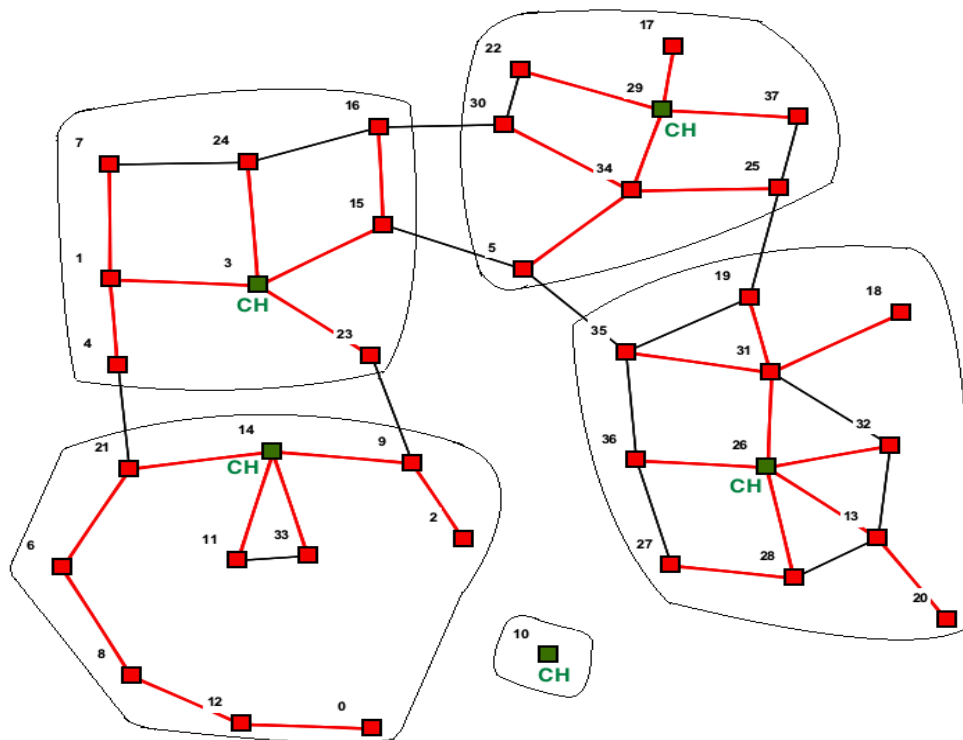


Figure 7.4: Dynamic cluster-based data aggregation

the cluster heads. In such topology of around 40 nodes, 5 disjoint clusters have been constructed.

7.3.2 Aggregated Data Transfer

Having defined the DICE protocol, let us move on to see how aggregated data are transferred to a sink, say S . We next describe how data is retrieved on-demand by the sink following the pull mode, and based on intensional destinations. Before proceeding with this section, we should note that in the following, we treat only the messages of type aggregation (also known as packet of type aggregation). The sink S floods a message M_i as an interest in the network asking all nodes that have aggregated data to send them to the sink. As we have seen in Section 7.3.1, a subset of sensor nodes is the cluster heads or has been considered as cluster heads (in dynamic topology). The message M_i is thus disseminated throughout the network with a destination specified intensionally, as shown in Figure 7.5(b). This dissemination sets up *gradients* with the network. Specifically, a gradient is created in each node that receives a message. The gradient is set towards the neighboring node from which the message is received, Figure 7.5(c). Aggregated data will be delivered towards the sink along eventually multiple gradient paths, Figure 7.5(d).

When receiving the message M_i by a node say γ , the *Reception module* transfers the message to the *Emission module* to propagate the message in the network, and at the same time transfers the payload of the message M_i to the *Engine* to be evaluated, as shown in Lines (4 – 5) in Algorithm 7.7. The payload of the message is a query that allows to aggregate the collected data using the average function, as shown in a particular representation in Listing 7.1. Any other function (e.g. median) to aggregate efficiently the collected data can be used.

7.3 Cluster-based Data Aggregation

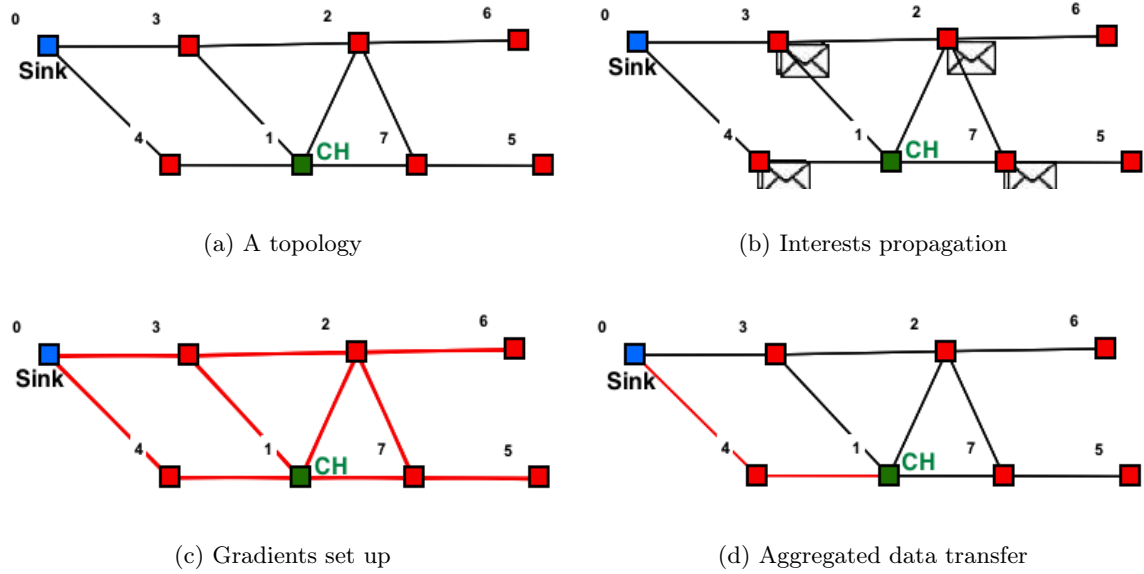


Figure 7.5: A simplified schematic for on-demand data transfer

```
SELECT avg(Data) FROM Aggregation;
```

Listing 7.1: Collected data aggregation

```

input: Message
1 foreach Message do
2   case Message.type is aggregation (packet of type aggregation)
3     // From sink to sensors (top down)
4     if Message.extDest is all then
5       Transfer Message to the Emission module;
6       Transfer Message.payload to the Engine;
7     else
8       if Message.extDest = sink-address then
9         // Node is the sink
10        Transfer Message.payload to the Engine;
11      else
12        // Transfer aggregation result towards the sink (bottom up)
13        Transfer Message to the Emission module;
14      end
15    end
16  end
17 end

```

Algorithm 7.7: Router Reception module: packet of type aggregation

It node γ has collected data, its *Engine* aggregates the data by evaluating the corresponding query, Listing 7.1. The computational result is encapsulated in a message and sent in unicast mode to the sink through multi-hop routing. More precisely, the *Engine* creates a message M_j , sets

7.3 Cluster-based Data Aggregation

up M_j 's payload by the computational result, M_j 's extensional destination by the address of the sink, and M_j 's intensional destination by an intensional query Q (described below). After that the *Engine* transfers the message M_j to the *Emission module* of the *Router*, as shown in Lines (3 – 6) in Algorithm 7.8. All nodes that have not been considered as cluster heads discard the message, Line (8) in Algorithm 7.8.

```

input: Message
1 foreach Message do
2   case Message.type is aggregation (packet of type aggregation)
3     Result  $\leftarrow$  Execute the aggregation query, Listing 7.1;
4     if Result is not empty then
5       Create new message Msg set Msg.payload = Result, Msg.extDest = sink-address,
        and Msg.intDest =  $Q$ ;
6       Transfer Msg to the Emission module;
7     else
8       Discard Message;
9     end
10 end

```

Algorithm 7.8: Functionalities of the Engine when receiving a packet of type aggregation

We have seen in Algorithm 7.5 how the *Emission module* routes messages that contain packets of type data. We next describe how this module routes messages that contain packets of type aggregation, Algorithm 7.9. We distinguish between two cases: (i) top down, when messages from sink to sensor nodes are disseminated, and (ii) bottom up, when aggregated data are sent from the cluster heads to the sink. In the top down case, the message is received by each node in the network, and each nodes builds different routes to the sink, Lines (4 – 6) in Algorithm 7.9. In the bottom up case, the *Emission module* for each message M_j gets the nexthop from the routing table and sends the message to the nexthop if a route exists, Lines (8 – 10) in Algorithm 7.9. Otherwise, it executes the intensional destination query Q to get a route to reach the sink, Line (12). The choice of the query Q depends on the application. It can be for instance a Questlog query that allows on-demand to find (the nexthop of) a route to the sink S . For instance, Q can be specified by the Questlog query $Route(\gamma, S, z, n)$ as seen in Section 5.1.2. As alternative solutions, Q can be specified by (i) a query that allows to find the nearest sink, or (ii) a random routing mechanism to get the (nearest) sink. If a new route is found, the message is routed as above. Otherwise, the message is discarded, as shown in Line (17) in Algorithm 7.9.

7.4 Experiments over QuestMonitor

```
input: Message
1 foreach Message do
2   switch Message do
3     case Message.type is aggregation (packet of type aggregation)
4       // From sink to sensors (top down)
5       if Message.extDest is all then
6         Send message to neighbors;
7         Store a route to the sink;
8       else
9         // From sensors to sink (bottom up)
10        Get nextHop to Message.extDest from routing table;
11        if nextHop is not empty then
12          Send Message to nextHop;
13        else
14          Result = Execute the intensional destination Q;
15          if Result is not empty then
16            Update Message set Message.extDest = Result;
17            Send Message to Result;
18          else
19            Discard Message;
20          end
21        end
22      end
23    endsw
24  end
```

Algorithm 7.9: Router Emission module: packet of type aggregation

When a node ψ receives the message M_j , the *Reception module* checks if node ψ is the sink to treat the aggregated data, as seen in Lines (7 – 8) in Algorithm 7.7. Otherwise, the message M_j is transferred to the *Emission module*, Line (10) in Algorithm 7.7.

7.4 Experiments over QuestMonitor

The proposed protocol, DICE, decomposes a network into a set of dynamic trees when data in messages with intensional destinations are traveling. Our protocol is inspired from the DDR [121] protocol in which the proof of a forest construction for any network has been proved.

We implement the DICE protocol using the Netlog language, and visualize its behavior on the QuestMonitor platform. We next visualize the behavior of the protocol in dynamic topologies to demonstrate its features such as load balancing and dynamic adaptation, show the characteristics of clusters as well as a particular case of the protocol.

7.4.1 Load Balancing

Figure 7.6(a) shows an example of a network topology with 28 nodes, while Figure 7.6(b) shows the generated trees after running the DICE protocol. Here we see four cluster heads elected on the fly, namely nodes 20, 16, 9, and 7. We want to visualize the behavior of the protocol upon topology

7.4 Experiments over QuestMonitor

changes. Figure 7.6(c) shows the resulting network topology after moving nodes 28 and 10. We notice that some of the cluster heads have been changed. Here we see that two of the previous four cluster heads are re-elected, and we obtain two new cluster heads, namely 1 and 12. We notice that some of the clusters reveal to be stable in regions with no perturbations. In dynamic regions, however, the DICE protocol adapts on the fly the clusters as well as the cluster heads. In Figure 7.6(d), we move node 12 and let two nodes 0 and 29 to join the network. In this case, we obtain four clusters with three new cluster heads, namely 7, 19, and 28. This allows to distribute the charge on different nodes. The dynamicity of clustering as well as the different cluster heads in dynamic topologies demonstrate that the DICE protocol provides high load balancing.

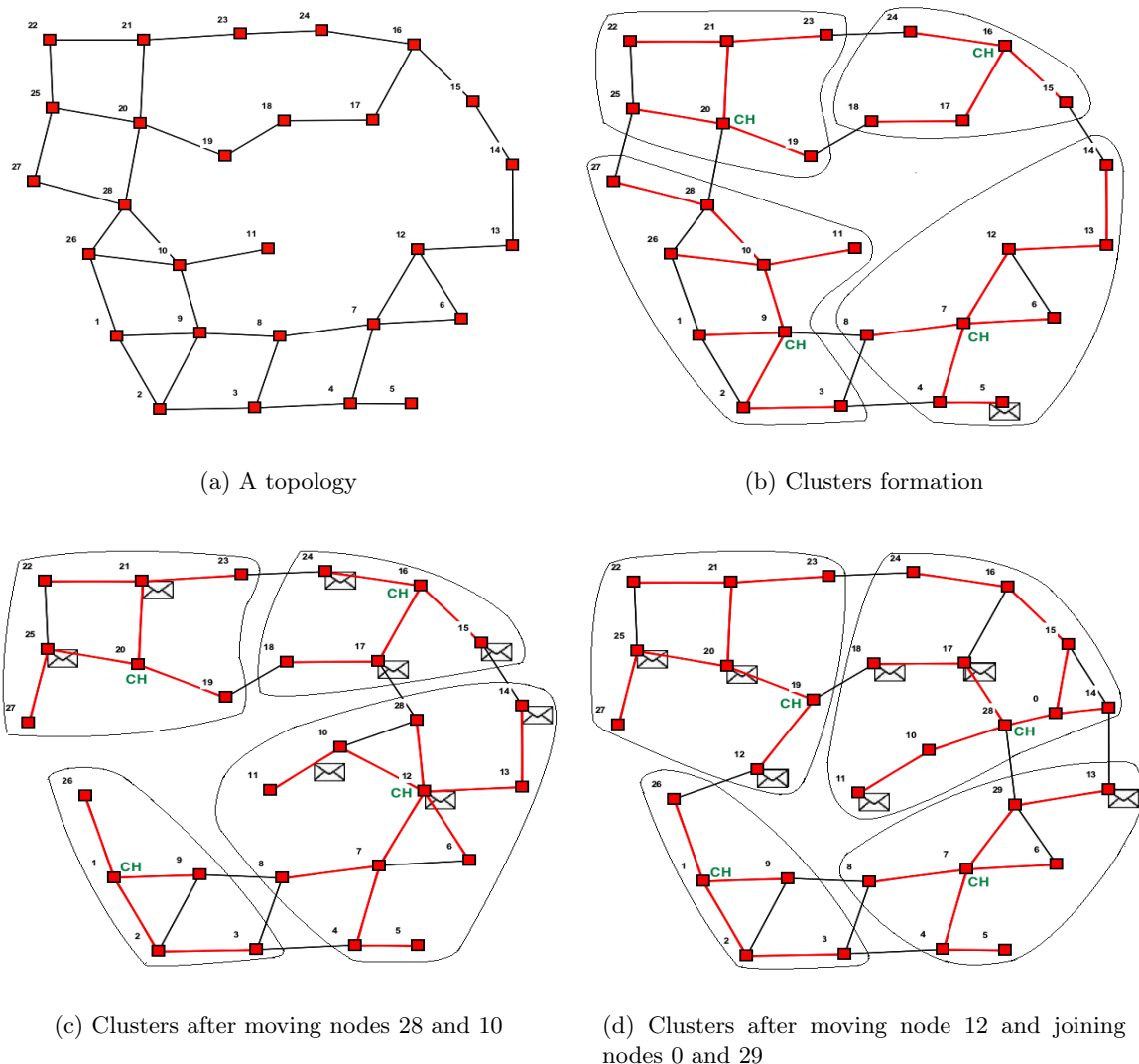


Figure 7.6: Dynamic adaptation of cluster heads upon topology changes

We have seen in Section 7.3 that the cluster heads store received data locally. Upon receiving a request from the sink, each cluster heads aggregates its data, and sends aggregated result to the sink.

7.4 Experiments over QuestMonitor

It is worthy to mention that in dynamic topology the cluster heads may be changed before receiving a request from the sink. In this case, those cluster heads should keep their data till receiving the sink's request. This request is disseminated with intensional destinations, with no prior knowledge of the identifiers of the cluster heads.

7.4.2 Dynamic Adaptation

We now demonstrate through an example how the DICE protocol adapts dynamically to topology changes. In Figure 7.7(a), we show a network topology where each node runs the DICE protocol. Each node transmits its sensed data to intensional cluster head specified on the fly when the message is traveling. At time t_x , each sensor node x transmits its sensed data. As a result, a set of trees are constructed on the fly. The network is decomposed into four clusters with four cluster heads namely 4, 8, 3, and 10. For instance, node 1 transmits its sensed data in a message m with intensional destination to a cluster head. As we have seen in Section 7.3, the intensional destination allows each node to choose a parent till the cluster head. The message travels nodes 15, 12, 20 and finally node 8 which is the cluster head.

At time t'_1 , the sensor node 1 retransmits its sensed data. However, the cluster head 8 now leaves the network, as shown in Figure 7.7(b). Node 1 then evaluates the intensional destination of the message m and chooses as parent, node 15, which repeats the same process and chooses as parent, node 12. Upon receiving the message m , node 12 evaluates the intensional destination, and gets as parent node 9. We notice that node 12 chooses another parent (which was node 20 as shown in Figure 7.7(a)) since the topology changes. Node 12 then sends the message to node 9. This node repeats the same process and chooses as parent node 10, which is the new cluster head. Figure 7.7(b) shows the resulting new clusters with their cluster heads, namely 4, 7, 10, and 3.

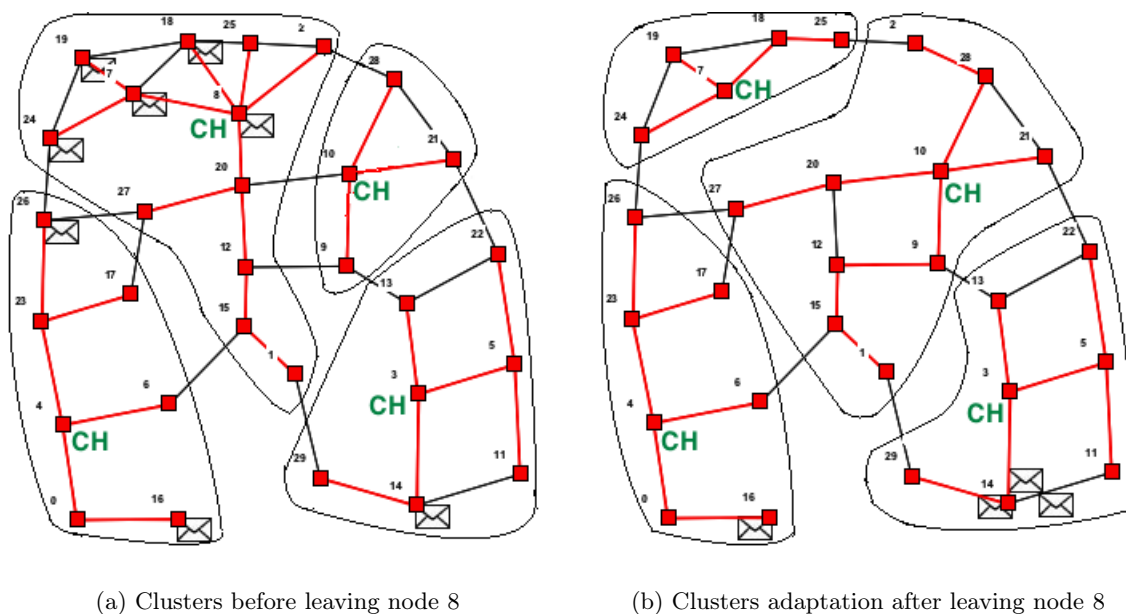


Figure 7.7: Dynamic clustering adaptation

7.4 Experiments over QuestMonitor

7.4.3 Characteristics of Clusters

We now study the characteristics of clusters such as the average number of cluster heads as well as the average number of sensor nodes per cluster. In Table 7.4, we recapitulate the principle parameters of simulation over QuestMonitor.

Parameters	Value
Nodes	20, 40, 60, 80, 100
Distribution	Random
Space	750 * 550
Transmission Range	100

Table 7.4: Parameters of simulation over QuestMonitor

We are interested to show the average number of cluster heads over various network topologies with different number of nodes distributed randomly. In Table 7.5, we show the resulting simulations. We notice that when the number of nodes increases, the number of cluster heads decreases till around 5 clusters. Let us take for instance the network topology of 20 nodes, the number of cluster heads is around 9.6, which means that approximately 50% of the nodes are considered as cluster heads. To clarify this result, let us recall that the nodes are randomly distributed in a space of 750 * 550. An increased number of nodes are scattered on the space, and consequently an increased number of clusters (cluster heads) are obtained. This result is illustrated as well by the resulting limited number of nodes per cluster, which is equal to 2.14, for the same topology.

Let us now to take the network topology of 100 nodes, we notice that the average number of cluster heads decreases, while the average number of nodes per clusters increases. If we increase the number of nodes, the degree of each node grows as well, and consequently an increased number of nodes per cluster are obtained. For instance, the average number of nodes per cluster for the network topology of 100 nodes is 17.85. This reduces the resulting number of clusters (cluster heads), which is equal to 5.7 for the same topology, as shown in Table 7.5.

Nodes	Average number of clusters	Average number of nodes per cluster
20	9.6	2.14
40	8.8	4.62
60	7.9	7.75
80	6.7	12.2
100	5.7	17.85

Table 7.5: Characteristics of clusters over QuestMonitor

7.4.4 Particular Case

There is a particular configuration where the protocol DICE provides trees with long branches. This configuration is when node identifiers are monotonically increasing or decreasing, as shown in

7.5 Discussion

Figure 7.8(a). In this case, all nodes forms one tree where node 1 is the root (cluster head), Figure 7.8(b). While this tree which might has very long branches is not optimal, the DICE protocol adapts dynamically to topology changes. Furthermore, this configuration is highly unlikely in a real world application, especially in dynamic topologies.

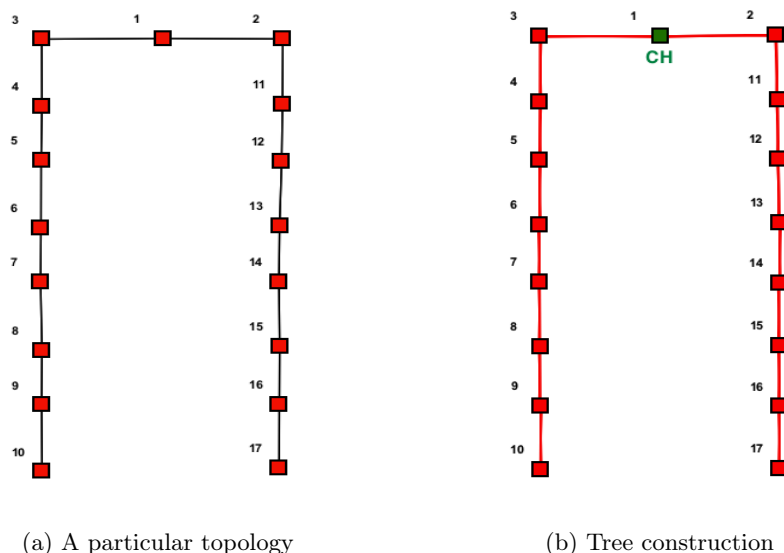


Figure 7.8: A particular topology

7.5 Discussion

The proposed protocol DICE decomposes a network into a set of dynamic clusters. This leads to supporting network scalability as well as to aggregating data efficiently. Thanks to the intensional destinations, each node calculates its parent on the fly by evaluating the intensional destination on its local data. The code of the intensional destination is mobile. In addition to the elegance of programming in active messages [156], the importance of this protocol is that it requires only knowledge of 1-hop neighbors. The communication complexity is local. This protocol is well-suited for dynamic networks. It adapts dynamically to topology changes. With respect to classical clustering approaches, this protocol results in limited communication overhead in dynamic topologies. However, it might have an increased cost in static topologies. Classical approaches set up clusters and elect cluster heads only once. Then each node sends its sensed data to the cluster head known *a priori*. In our case, however, there is no set up phase to make clusters and elect cluster heads, but the clusters are dynamically created, and each node needs to evaluate the intensional destination of the received message in order to calculate its parent, which might be the cluster head. This results in an increased local computation complexity.

In the DICE protocol, each node implicitly maintains its parent. When a node evaluates the intensional destination to get a parent, it updates the local data structure *Parent*. Note that we can add functionality to allow each node to support as well its children, without increased overhead. This can be done by adding a local data structure *Child* for instance, and allowing each node upon receiving a message to store the identifier of the expeditor node (received in the message) in the table

7.5 Discussion

Child. We need to maintain and update the table *Child*. In our protocol, each node periodically sends *Hello* message to neighbors. This is used to update the table *Neighbor* on each node. But it can be used as well to update the table *Child*. A node deletes its child x if it does not receive from x a *Hello* message for p ($p = 2$ for instance) periods. As a result, each node implicitly acquires information (parent and children), which can be useful for routing in intra-clusters.

More importantly, sensed data travel in messages with intensional destinations to (unknown) cluster heads, which are discovered on the fly. We have seen in Section 7.1 that each sensor node can periodically send its collected data to the sink. Sensor nodes are not synchronized to send their sensed data on the same time. But, we can benefit from the tree constructed on the fly, and allow each parent node to aggregate the received data before sending them to its parent. This can be done by letting all nodes that received messages with type data, to wait a time Δt before proceeding. If the node receives other data, it aggregates all received data, and sends the aggregated result to its parent calculated on the fly. In this way, we can decrease the number of messages propagated intensionally to the cluster heads.

We have seen in Section 7.3.2, that aggregated data are collected periodically on-demand by the sink. We can use another strategy. Instead of sending periodically queries by the sink, we allow the sink to build a tree, where the sink is the root of the tree. Each sensor node chooses a parent and updates it periodically by benefiting from the *Hello* messages between 1-hop nodes exchanged regularly. We suppose that the cluster heads store received sensed data for a time t . When t expires, the cluster heads aggregate their collected data and send aggregated results in the *push* mode to the sink following the tree.

Finally, we implemented the protocol DICE using the Netlog language on the Netquest virtual machine, and visualized its behavior on the QuestMonitor platform. However, this implementation does not allow to study the performance of the protocol. In the near future, we plan to conduct a set of simulations over a real simulator such as WSNNet to evaluate the performance of the protocol and carry out an analysis to make a comparison between DICE and some other protocols.

Conclusion

In this chapter, we developed declarative programs expressed in the Questlog language to collect, aggregate and transfer data towards sink nodes. The communication is based on messages whose destination is specified intensionally with Questlog queries. We then introduced a clustering protocol that allows to construct dynamic trees as clusters and aggregate data in a simple, modular and efficient manner. The cluster heads are elected on the fly when sensed data in messages are traveling. We implemented the DICE protocol on the Netquest virtual machine and demonstrated through examples over the QuestMonitor platform. We showed that the DICE protocol based on intensional destinations provides persistence to data traveling in the network, offers high load balancing, and adapts dynamically to topology changes.

Ubiquest/Netquest Systems and Experiments

8

Contents

Introduction	131
8.1 Ubiquest System	131
8.1.1 Data Structures and Languages	132
8.1.2 Ubiquest API	134
8.1.3 Ubiquest Engines	135
8.1.4 Local BMS	136
8.2 Experimentation and Validation	137
8.2.1 Ubiquest Simulation Platform	137
8.2.2 The Results	145
Conclusion	149

8.1 Ubiquest System

Introduction

Our works that we proposed in this thesis have been integrated into an ANR project UBIQUEST, which involves the three research laboratories CITI - INSA Lyon, LIG - Grenoble, and LIAMA - Beijing. Ubiquest [134] is a system that allows rapid prototyping of networking applications. It is composed of three main modules: (i) an Ubiquest Engine, (ii) an application programming interface (API), and (iii) a local Data Management System (DMS). The Ubiquest Engine contains sub-engines (e.g. Distributed Query Engine, Netlog and Questlog Engines) to evaluate SQL-like queries, to execute rule-based programs, and to maintain sensed data and physical neighbors. The API allows to exchange data and queries between nodes, while the local DMS manages application and network data, as well as additional information for distributed query evaluation.

Modeling networking protocols using declarative queries has already been demonstrated [94, 96, 98, 66]. In particular, Netquest team-project in which I am a member, provides an approach that allows to develop concise network programs based on the rule-based languages Netlog [66] and Questlog. Netquest offers as well a visualization tool, QuestMonitor [29], to monitor the behavior of nodes in a network and interact with the nodes at run time.

The Ubiquest approach, however, goes one step further by mixing in the same model, query language data management and network management. The network management is handled by the Netquest system which has been included in the Ubiquest system. While the data management is handled by a distributed query engine (DQE) designed to manipulate global SQL-like queries expressed in a query language called Data Location Aware Query Language (DLAQL) [134, 10]. The DQE engine as well as the query language DLAQL are managed by a team (*HADAS*) of LIG. An important characteristic of the Ubiquest approach is the possible interaction between DQE and rule-based engines. In particular, a query plan is generated for each DLAQL (sub)query with the possibility to invoke a rule-based program to evaluate a (sub)query.

The Ubiquest system proposes as well a simulation platform, which is an extension of the QuestMonitor [29] visualization tool, in order to monitor the behavior of network programs, and to realize some measurements such as the cost of executing a query plan. We participated as well to the design of the Ubiquest simulation platform [13].

Our contributions on the Ubiquest system are summarized as follows. We ported the Netlog Engine [66] as well as the Netlog compiler on the Ubiquest system. We implemented the rule-based Engine Questlog and the Questlog compiler, which have been defined in Chapters 5 and 6. We contributed also with some colleagues on the implementation of the modules of the Ubiquest system [7, 8] (specifically reception, emission, payload dispatcher, and communication modules shown in Figure 8.2), and on the design of the Ubiquest simulation platform [13].

In this chapter, we present only the architecture of the Ubiquest system and the Ubiquest simulation platform. For more information about the DLAQL language as well as the DQE engine, please refer to [110, 134].

The chapter is organized as follows. In the next section, we present the architecture of the Ubiquest system. In Section 8.2, we describe the Ubiquest simulation platform, and present some experiments to monitor the behavior of declarative network programs.

8.1 Ubiquest System

Ubiquest is a data-centric approach that provides a unified view of "objects" handled by both the networks and applications. Nodes in a network communicate by exchanging messages and

8.1 Ubiquest System

interact through declarative queries including rule-based programs (e.g. routing) and/or specific data-oriented distributed algorithms (e.g. distributed join).

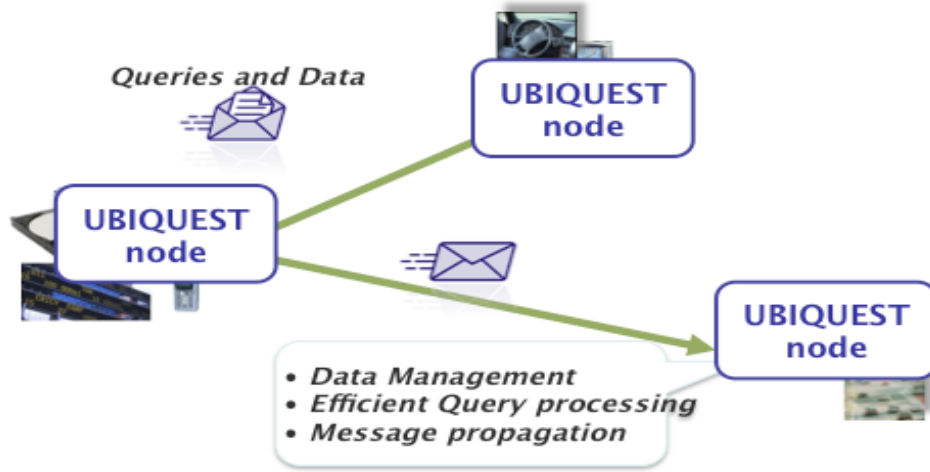


Figure 8.1: An Ubiquest system

An Ubiquest system runs on a set of devices (nodes) interconnected through a wireless network (Figure 8.1). Every device embeds a virtual machine in charge of data management, processing queries (data selection and updates) and messages propagation. The Ubiquest virtual machine (VM) as shown in Figure 8.2 is composed of: (i) a local DMS to manage application data, network data and additional information for distributed query evaluation; (ii) an Ubiquest Engine comprising sub-engines in charge of evaluating queries, executing rule-based programs, maintaining sensed data and the list of physical neighbors, and (iii) application programming interfaces (APIs) with different modules to exchange queries and data in messages between nodes. An Ubiquest node is a device equipped with a VM complemented with a device wrapper that allows device/VM interaction.

All exchanges between nodes are carried out by queries and data in messages. Queries are defined using either rule-based languages (Netlog or Questlog) for network data and query expressions, or declarative query language (DLAQL) for querying application data with a global point of view. DLAQL queries are optimized based on a case-based reasoning (CBR) approach and pseudo-random query plan generation [110]. We next present the data structures and languages used by Ubiquest, and afterwards we present the components of the Ubiquest virtual machine.

8.1.1 Data Structures and Languages

In this section, we present the data structures, the data distribution, the message structure, and the declarative languages used by the Ubiquest system.

Item and Itemsets

Network and application data in Ubiquest are Itemsets. An item is the unit of data manipulation: rules (in programs) are evaluated for each new item (new fact), and a query is processed Item by Item following the classical iterator model. An Item is composed of a set of attribute/value couples, values are taken in predefined data types including integer, float, string, date and *NodeId* (node identifier type). The predefined attribute *LocalID* value (of type *NodeId*) is the identifier of the

8.1 Ubiquest System

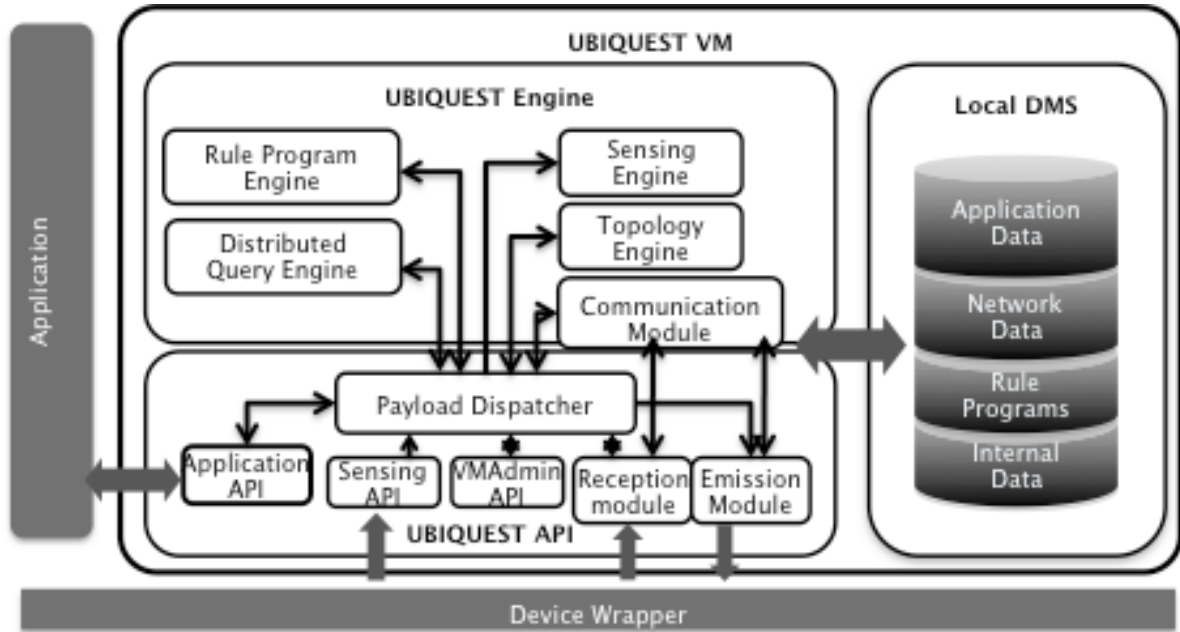


Figure 8.2: Ubiquest virtual machine

current node. An Itemset is a set defined by a name and the structure shared of its items, e.g. a set of attributes (name, type). Key attributes are used to identify one specific item.

Data distribution

Ubiquest supports only horizontal fragmentation of Itemsets. This means that a global Itemset is distributed over several (maybe all) Ubiquest nodes. Any participant node stores a (local) Itemset or a fragment with the same schema as the global one. Each item of the global Itemset is actually stored in one or more nodes. Data distribution is application-driven: applications decide on which node(s) items have to be stored. There is one exception to this rule: Items using LocalID as one of its key attributes are stored on the node corresponding to the LocalID value. For example, the global Itemset describing the physical communication links in the network has type:

$$Link(LocalID\{key\}, NeighbourId\{key\})$$

It is distributed according to the LocalID value. That means that each node stores its neighbors.

Message structure

A message (Figure 8.3) is the unit of communication among nodes. A message has two main parts: (i) a networking information, and (ii) a payload where the content of the message (e.g. queries or items) is embedded.

The networking information may contain (i) a logical destination of the message defined as a set of expressions of type NodeId. An expression is either a NodeId value (extensional expression) or a query returning NodeIds (intentional expression); (ii) an Immediate destinations of the message, e.g. a set of neighboring NodeId values, (iii) a ProgramId identifying the program used for the

8.1 Ubiquest System

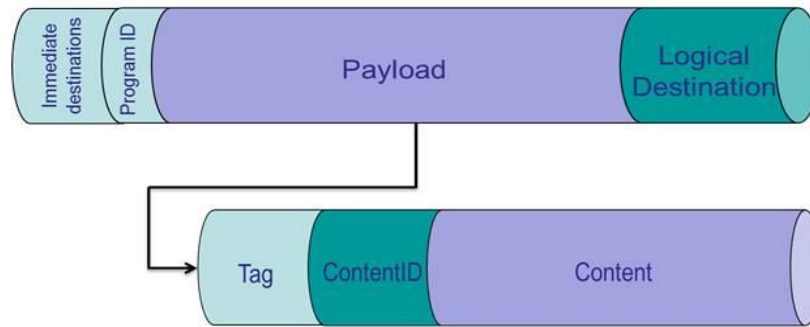


Figure 8.3: Ubiquest Message structure

propagation of the message in the network. For instance, a message from node A to node D may pass through nodes B and C. The networking information part of the message emitted from the intermediate node B is:

- Logical destinations: D
- Immediate destinations: C
- ProgramID, the identifier of the message received by Node B (same networking protocol).

The payload has three parts: (i) a tag identifying the type of content (e.g. declarative query, query results, data/facts); (ii) a contentId identifying in a unique way the content; (iii) the content which can be either declarative queries or data (query results or facts).

Declarative languages

The Ubiquest system handles rule-based languages as well as an SQL-like query language. The Questlog language is described in Chapter 5, and Netlog is overviewed in Chapter 4. Data Location Aware Query Language (DLAQL) is a global SQL-like query language defined by Ubiquest. DLAQL extends the well-known SQL2 data manipulation language to conform to the data distribution policy of Ubiquest. This means that a DLAQL expression may explicitly indicate on which Ubiquest node data has to be stored to or deleted from. For more information about the DLAQL language, please refer to [10].

8.1.2 Ubiquest API

The Ubiquest API manages all interactions between the Ubiquest Engines and local applications, device sensors as well as other VM through message exchange. As shown in Figure 8.2, the API is composed of: (i) Application API, in charge of the interaction with applications running on the local node, (ii) Reception and Emission modules to deal with message exchange among Ubiquest nodes, (iii) Sensing API that locally stores data coming from sensors embedded in the physical device, and (iv) Payload Dispatcher, which manages Payload exchange among VM sub-components.

- The *Application API* module validates DLAQL queries/updates submitted by applications, and translates them into an internal representation before sending them to the Ubiquest Engine for evaluation.
- The *Reception Module* receives messages from other Ubiquest nodes and decides if the payload

8.1 Ubiquest System

of the incoming message has to be treated locally. It checks if the local node is part of the logical destination of the message. This process may involve interaction with the Ubiquest Engine to resolve intentional expressions of logical destinations (e.g. destination expressed using a query). Finally, the Reception Module sends the Payload of the message to the Payload Dispatcher to treat it if the local node is one of the destinations, and otherwise forwards the message to the Emission Module.

- The *Emission Module* builds a new message using Payload, logical destinations and a programId identifying a dissemination protocol, and invokes the Ubiquest Engine to compute the immediate physical destination(s) from the logical ones. Finally, it sends the message over the network using a program (routing protocol) selected by the Communication Module, which will be described later.
- The *Payload Dispatcher* maintains a record of the identifiers of payloads that are currently executed at the node. This allows determining if a received payload was already executed, and thus avoids loops. When it receives payloads from the Application API, it generates a new identifier for registering. When it receives payloads from the Reception Module, the Payload is forwarded to the corresponding Engine for treatment.

When a payload's identifier is not in the record, the Payload Dispatcher generates and registers a new identifier, and transfers the payload to the corresponding Engine. Otherwise, the identifier exists in the records, then the payload dispatcher transfers the payload to the corresponding Engine according to the query/result type and the payload identifier.

The payload dispatcher receives as well messages from the Ubiquest Engine. These messages are transferred to the Emission Module to be sent over the network.

8.1.3 Ubiquest Engines

The Ubiquest system contains two main engines to evaluate (i) DLAQL queries and (ii) rule-based programs, as well as two simple engines to monitor and update topology and physical sensors measurements.

Netlog/Questlog Integration

The *Rule Program Engine* is in charge of executing rule-based declarative programs exploited for specifying distributed algorithms (e.g. networking protocols, sub-query execution). The Netlog engine has been described in Chapter 4, and the Questlog engine has been defined in Chapter 6.

DQE Engine

The *Distributed Query Engine* (DQE) is responsible of executing DLAQL queries. The role of the DQE Engine is to build and execute efficient local query execution plans according to a given cost function (expressed as a combination of real cost parameters). Execution plans are composed of classical physical operators (implementing algebraic operations) and specific operators to invoke program or propagate subqueries, as shown in Figure 8.4.

8.1 Ubiquest System

A query plan is a tree whose root node corresponds to a DLAQL command (e.g. SELECT, INSERT, DELETE, UPDATE), intermediate nodes correspond to computation operators (e.g. unions, joins, filter or aggregate), and leafs correspond to data access operators: local DMS querying, sub-query emission to neighbors, or rule-based program invocation. Efficient execution plans are selected using a combination of Case-Based Reasoning and pseudo-random query plan generation. For more information about the DQE engine, please refer to [110, 9].

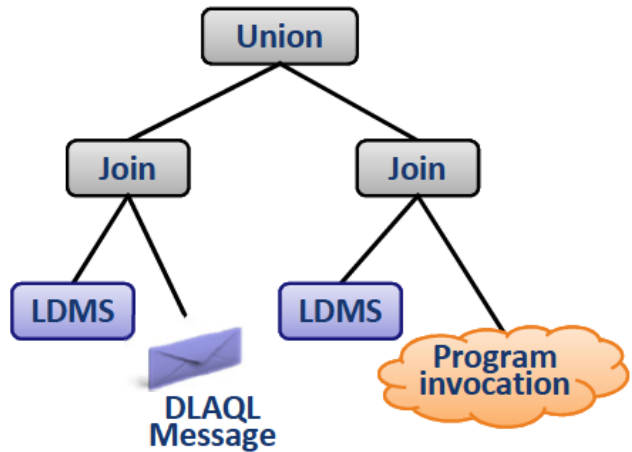


Figure 8.4: A query plan

Sensing and Topology Engines

These two modules are autonomous and react to changes in the environment detected by the device and signaled through the Device Wrapper. The *Sensing Engine* gets the measures coming from physical sensors embedded in the device (e.g. temperature, location) and stores these values in corresponding itemsets. These itemsets are predefined. They adopt a common structure (e.g. itemset Temperature(NodeId {key}, value)). The *Topology Engine* is responsible of updating the Link itemset, defined as Link(NodeId {key}, Neighbor {key}) according to physical network connections that are established or removed. The Link itemset is mandatory and is sufficient to permit communication among nodes.

Communication Module

The *Communication Module* has two different roles: (i) determine if the local node is part of the logical destination of incoming messages, and (ii) determine what is the next hop to transmit a message to a logical destination. The logical destination of a message is either expressed extensionally using a list of node identifiers, expressed intentionally using a query returning node identifiers, or expressed by a combination of both. If it is expressed extensionally, determining if the local node takes part in the logical destination of a message is straightforward. In the other case, the Communication Module asks the DQE Engine to solve the intentional destination (e.g. obtain extensional destinations) before deciding. To determine the next hop(s) for propagating a message, the Communication Module selects a propagation program and invokes the appropriate rule Program Engine to execute it. The default propagation program simply do broadcasting to all neighbors (e.g. the next hops correspond to all items of the Link itemset). Other propagation programs can be written by developers (e.g. by exploiting and maintaining a routing table) and may be automatically selected by the Communication Module.

8.1.4 Local BMS

The local DMS stores and manages data as Itemsets. As shown in Figure 8.5, it contains application data (e.g. sensed data), network data (e.g. routing tables, neighbor able), rule-programs (e.g.

8.2 Experimentation and Validation

distributed algorithms that can be dynamically loaded/removed to/from the system), and internal data (e.g. device specific data) used for running other Ubiquest VM components.

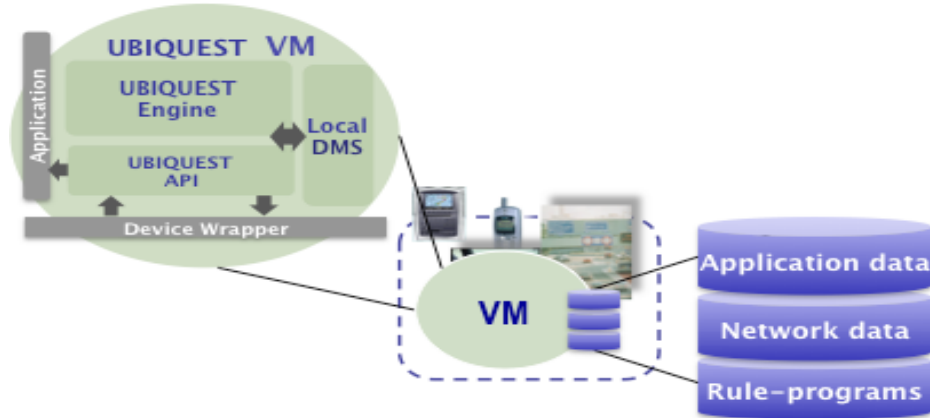


Figure 8.5: Ubiquest node components

8.2 Experimentation and Validation

In this section, we present the architecture of the Ubiquest Simulation platform and monitor the behavior of protocols, expressed by the Netlog and Questlog languages, through the Ubiquest simulation platform.

8.2.1 Ubiquest Simulation Platform

The Ubiquest Simulation platform (Figure 8.6) allows to simulate a network of Ubiquest nodes. This is achieved by running a set of Ubiquest virtual machines. This platform extends the QuestMonitor visualization tool [29] that has been used with the Netquest [66] system. In contrast to QuestMonitor which has been designed to run with the Netquest virtual machine, the Ubiquest simulation platform is independent from the Ubiquest virtual machine. It can be used with any other system to visualize databases.

This platform allows to visualize dynamic networks, monitor the execution of protocols and queries, and interact with nodes in the network. The objective is to monitor the behavior of distributed protocols and applications.

The Ubiquest simulation platform has: (i) a code editor interface that helps developers to write rule programs, and (ii) a simulation interface that has four main components:

- The Network Parameters Window (*Program management*), which allows to set up a network with various groups of nodes;
- The Network Graphical Window (*Network monitor*), which allows to visualize and interact with the nodes at run time;
- The Logs Window (*Logs monitor*), which displays the log of a given node;
- The Node Settings Tabs (*Node*), which allows to interact with a selected node.

8.2 Experimentation and Validation

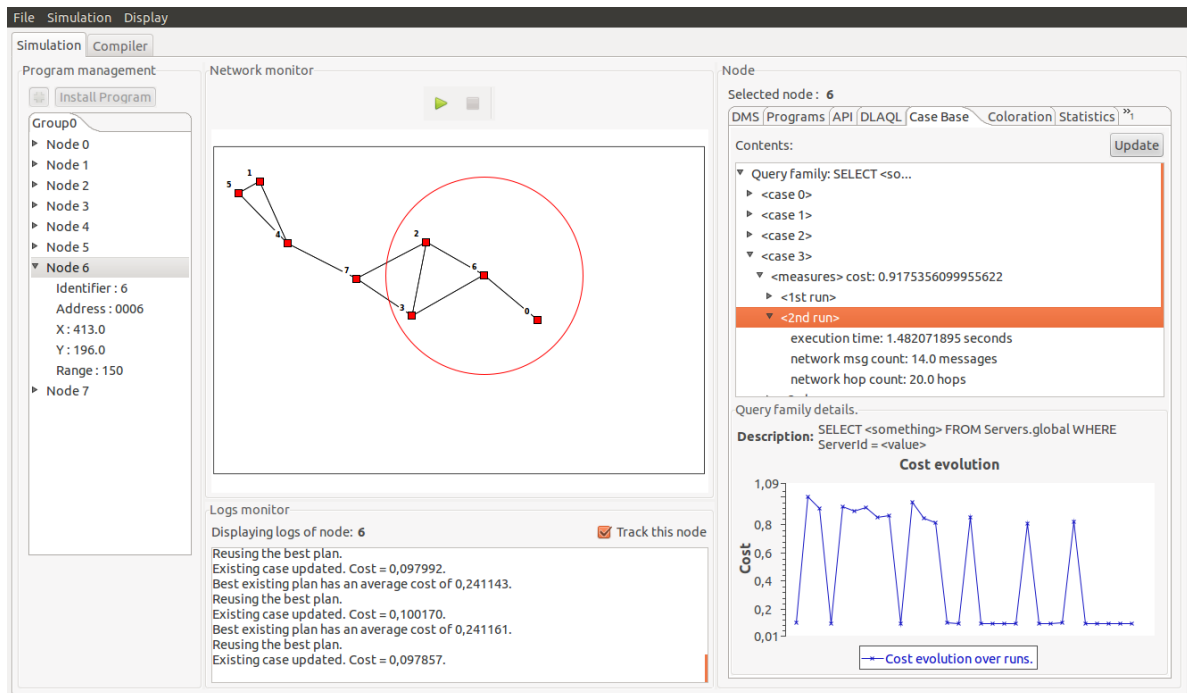


Figure 8.6: Ubiquest simulation platform user interface

Network Parameters Window

The Network Parameters Window allows to create groups of nodes (Figure 8.7), to display the status of the nodes (Figure 8.8), and to install rule-based programs on them (Figure 8.9). Different groups of nodes in a network can be created. The different groups can have different colors and radio range, and can comprise mobile or fixed nodes. These settings are validated by pressing the *Draw graph* button. At this time, the system creates the groups of nodes randomly.

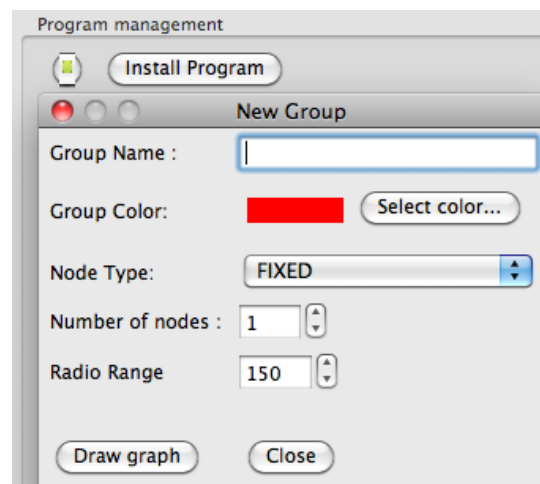


Figure 8.7: Create group of nodes

Upon creation, the group of nodes can be displayed. For each node, its identifier, address, position and radio range can be shown. In order to finalize the creation of a group, the data-centric protocols have to be installed on each node. The system allows to install several protocols on the nodes.

8.2 Experimentation and Validation

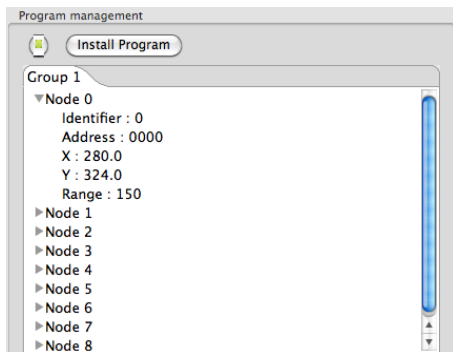


Figure 8.8: Display nodes

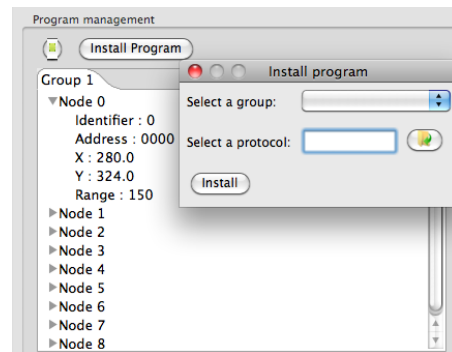


Figure 8.9: Install rule-based program

Network Graphical Window

The Network Graphical Window (Figure 8.10) offers the view of the different groups of nodes represented by different shapes and possibly different colors, as well as the connections between them (if the nodes are located inside the radio range of another node). Each node is identified by a unique address.

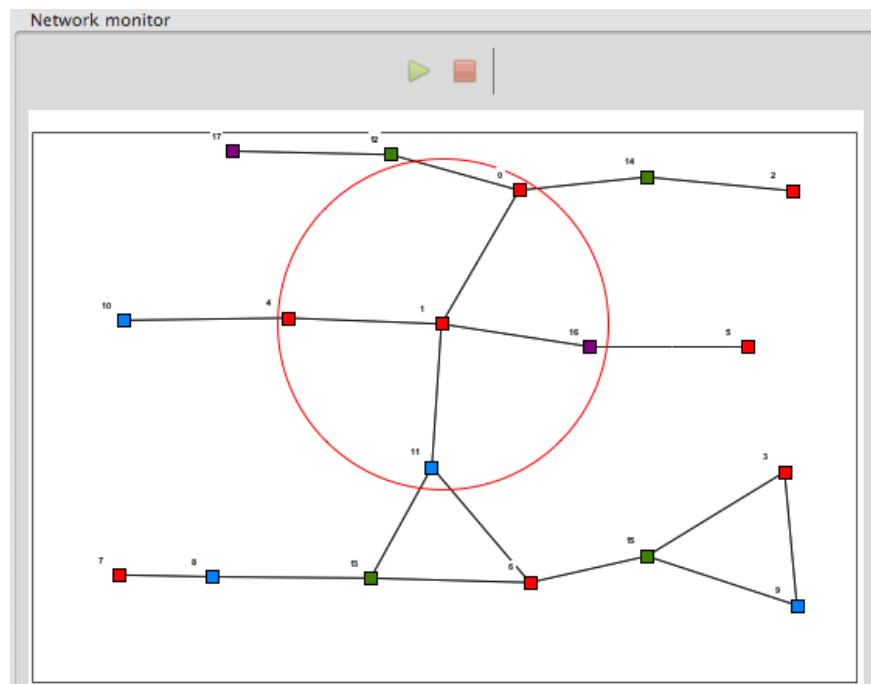


Figure 8.10: Network graphical window

The Network Graphical Window also allows to interact with the network, and modify its configuration by moving nodes, deleting edges or nodes for instance. This part was implemented using Piccolo2d¹ which allows to create Zoomable User Interfaces (ZUIs). The user can smoothly zoom in or out in order to get more details or have a global overview of the network (pressing Ctrl + right click and moving the mouse to the right or the left). Piccolo2d has also a hierarchical structure which permits to manipulate easily a group of identical objects: edges, messages, ranges, nodes

¹<http://piccolo2d.org/>

8.2 Experimentation and Validation

identifiers, etc. The users can thus easily display the layers they are interested in without being disturbed by too many information (especially for big networks).

Logs Window

The Logs Window (Figure 8.11) displays the log generated by a given node. This log is updated at run time. By default, the log shows the information about the node selected in the Graphical Window.

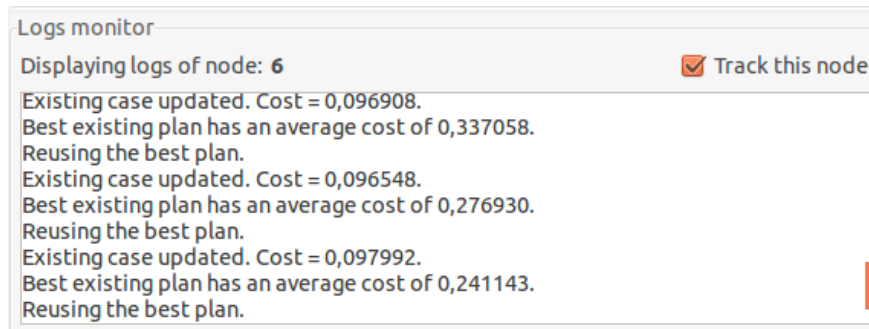


Figure 8.11: Logs window

Sometimes it is desirable to view the log of a given node while interacting with another one. For instance, a user selects node 6 to view its logs. Then, the user wants to issue a global DLAQL query on node 4, but still following the log of node 6. This can be achieved by clicking on the checkbox "track this node" on the upper right window, as shown in Figure 8.11, before selecting node 4 on the Graphical Window.

Node Settings Tabs

The Node Settings Tabs exhibits informations about the node selected by the user, displayed on the right side of the user interface in Figure 8.6. For instance, a user can choose a node and visualize the data in the local data store, send a fact or a query in the network, see received and sent messages and show their contents, check the programs installed on the node, etc. This window contains the following tabs: DMS, Programs, API, DLAQL, Case Base, Coloration, Statistics, and Messages.

The *DMS* tab (Figure 8.12) allows the user to view the content of the local database stored at the selected node. For example, one can choose to display the content of the table *Route* to check existing routes on the selected node. Figure 8.12 displays the routing table on node 6.

dt	nexthop	nbhops	routesn	exptime
0000	0000	1	28	1021737513
0001	0002	4	22	1021737488
0002	0002	1	28	1021737488
0003	0003	1	28	1021737462
0004	0002	3	24	1021737488
0005	0002	4	22	1021737488
0006	0006	0	30	0
0007	0002	2	26	1021737488

Figure 8.12: DMS tab

8.2 Experimentation and Validation

The *Programs* tab (Figure 8.13) displays the protocols that are installed on the selected node. The user is allowed to enable or disable them on this node.

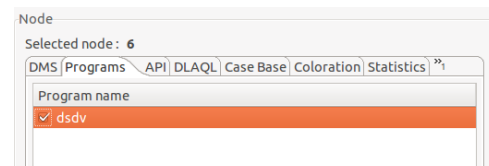


Figure 8.13: Programs tab

The *API* tab (Figure 8.14) allows to interact with the Netlog/Questlog engine of the selected node by adding for instance a fact in one of the tables of the nodes, as an application would do, by updating some sensed data for instance. It can be used also to fire reactively a Questlog query at run time. Since different programs can be installed and run on the same node, the appropriate program should be selected. As shown in Figure 8.14, the query/fact with the related program and the timestamp will be displayed when an answer is received.

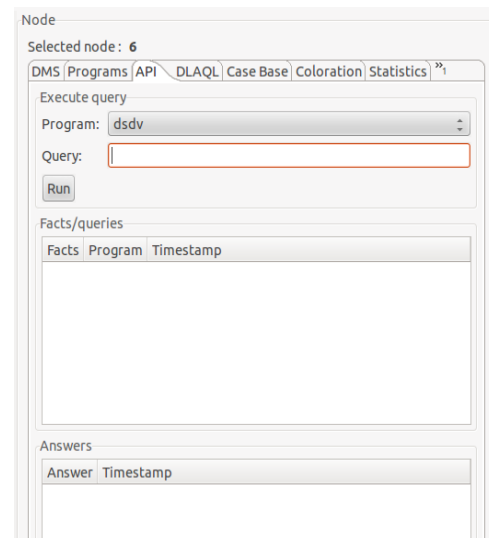


Figure 8.14: API tab

The *DLAQL* tab (Figure 8.15) allows to type DLAQL queries and to submit them to the selected node. Figure 8.15 shows a DLAQL query submitted to the related engine of node 6. Once executed, the results are displayed.

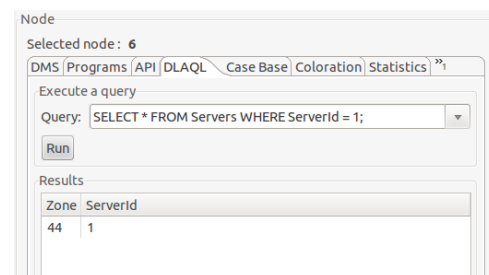


Figure 8.15: DLAQL tab

8.2 Experimentation and Validation

The *Case Base* tab (Figure 8.16) displays the content of the case base and information about the learning process. The query families processed at the selected node are listed at the top of this tab, with details about the performance of each case (e.g. query plan). The query plan is also represented graphically, as a tree of operators. For each part, some details are displayed for clarity. For example, when a query family is selected, a SQL-like description of this query family is provided, together with a chart showing how the cost of query executions evolved over time.

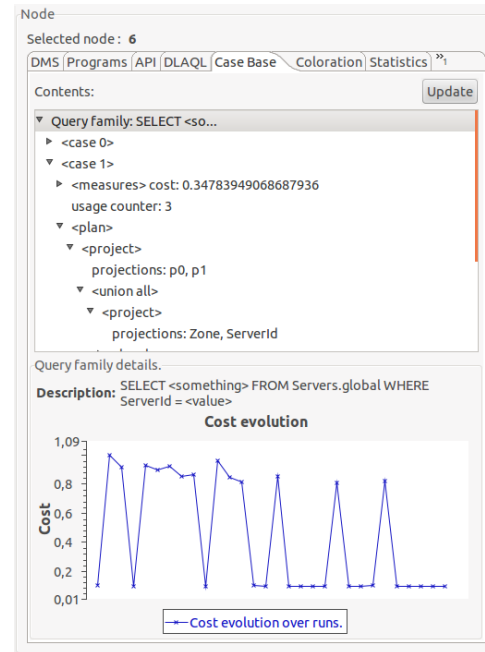


Figure 8.16: Case Base tab

The *Coloration* tab (Figure 8.18) allows the user to color edges in order to render a given information in a visual way. For example, it allows to draw a network path as shown in Figure 8.17. This tab is mainly useful for network protocol developers.

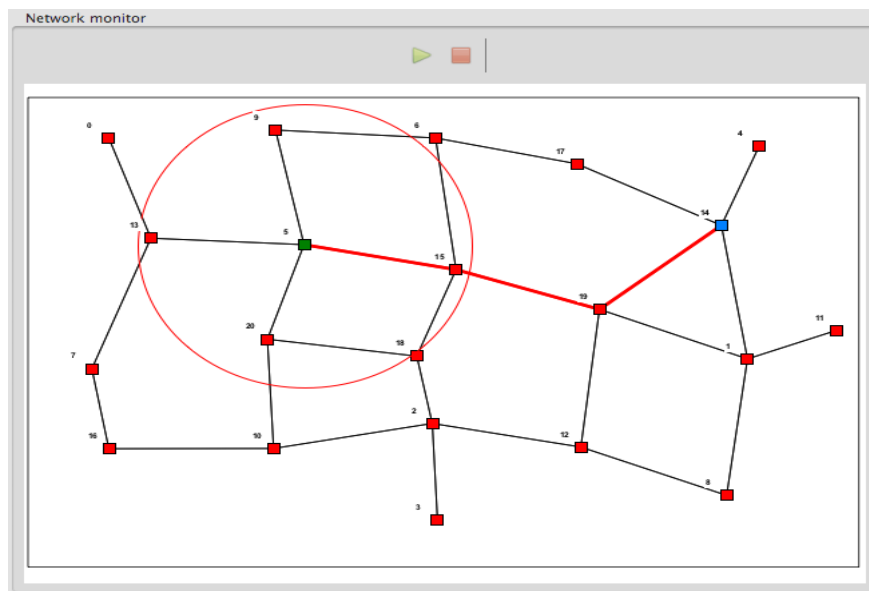


Figure 8.17: Coloring a network path

The QuestMonitor platform offers four distinct levels of coloration: (i) local, (ii) recursive, (iii) path, or (iv) global.

8.2 Experimentation and Validation

- **Local:** The user selects a table and one of its attributes. The system checks in the selected table, if for one of the tuples, the selected attribute corresponds to one of the neighbors of the selected node and color the edge between them.
- **Recursive:** The user must select two attributes, and the system will check each of these attributes in the same way as in the *local* case. The first attribute infers the same coloration, whereas the second attributes is interpreted differently: the matching set of neighbors is recorded, and these operations are repeated recursively to each of them.
- **Path:** The user selects two attributes as well as source and destination nodes. The system tries to find a path from the source to the destination using the *Recursive* method.
- **Global:** This case is equivalent to the *recursive* case applied to all the nodes (not only the one selected).

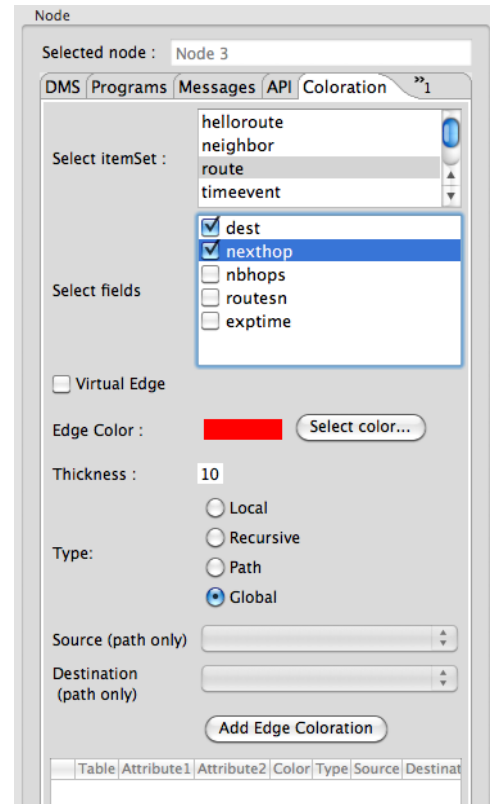


Figure 8.18: Coloration tab

The *Statistics* tab (Figure 8.19) shows some basic statistics about the node in order to measure the complexity in communication and computation. Such statistics concern for instance the number of select and update queries, as well as the total number of queries executed in the database of the selected node.

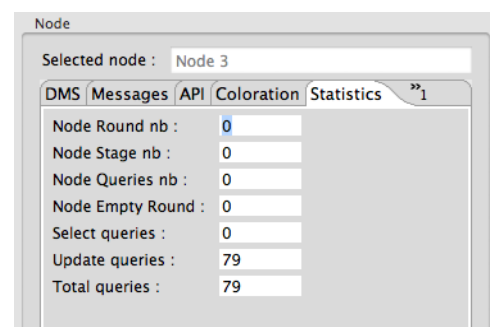
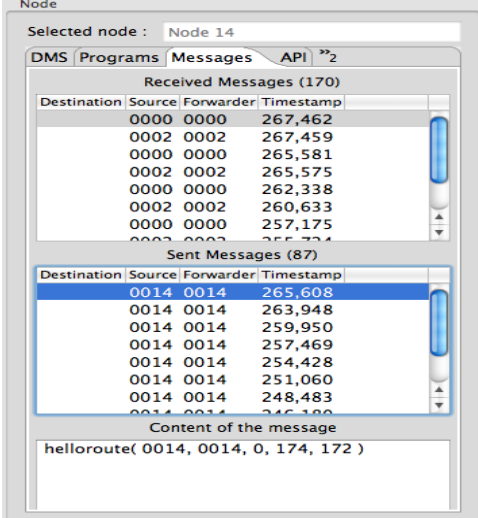


Figure 8.19: Statistics tab

8.2 Experimentation and Validation

The *Messages* tab (Figure 8.20) displays all the messages received or sent by the node. As shown in Figure 8.20, for each message received or sent, the destination of the message as well as the source node, the forwarder (intermediate node), and the timestamp (time upon received or sent a message) are displayed. If the user clicks on one message in particular, the content of the message which can be a fact, a query, or an answer is displayed.



Node
Selected node : Node 14

DMS Programs Messages API 2

Received Messages (170)

Destination	Source	Forwarder	Timestamp
0000	0000		267,462
0002	0002		267,459
0000	0000		265,581
0002	0002		265,575
0000	0000		262,338
0002	0002		260,633
0000	0000		257,175
0002	0002		255,734

Sent Messages (87)

Destination	Source	Forwarder	Timestamp
0014	0014		265,608
0014	0014		263,948
0014	0014		259,950
0014	0014		257,469
0014	0014		254,428
0014	0014		251,060
0014	0014		248,483
0014	0014		246,180

Content of the message
helloroute(0014, 0014, 0, 174, 172)

Figure 8.20: Messages tab

Code Editor and Compiler

The code editor (Figure 8.21) is an environment for helping developers to write rule programs using either Netlog [66] or Questlog as we have seen in Chapter 6. It provides functionalities such as syntax coloring and error detection. It transforms a rule program into a file containing a set of SQL expressions. This file needs to be installed on each node of a network before starting the simulation.

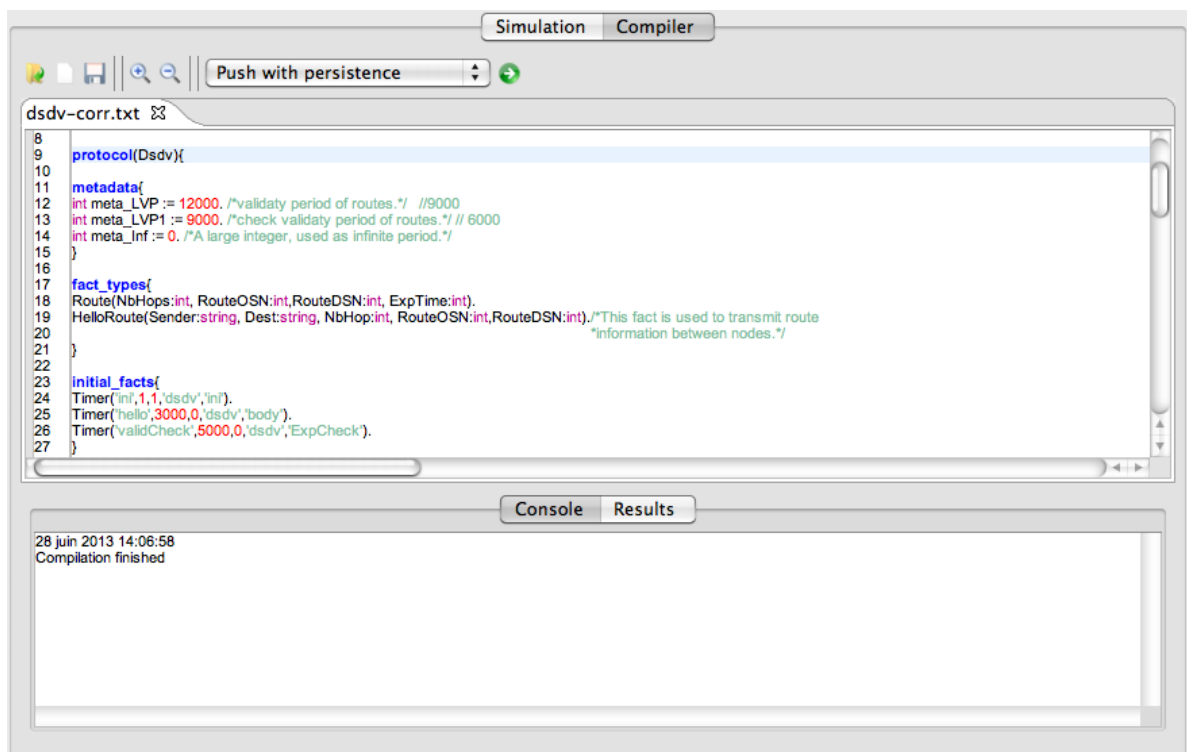


Figure 8.21: Rule programs code editor interface

8.2 Experimentation and Validation

8.2.2 The Results

We next present different examples of protocols expressed by the Netlog and Questlog languages, and simulate and monitor their behavior through the Ubiquest simulation platform.

DSDV Route

In Section 4.4.3, we presented the implementation of a simplified version of the DSDV protocol [126]. Using the Ubiquest simulation platform, we can easily simulate, test, and debug the implementation of this protocol.

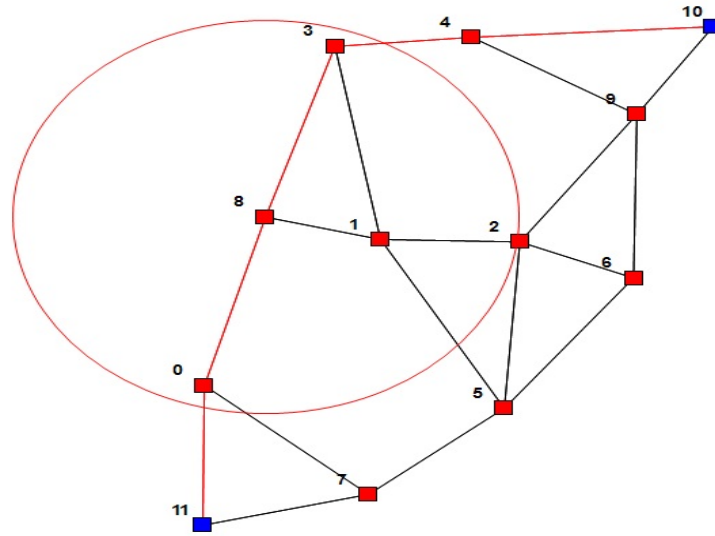


Figure 8.22: A network with DSDV

Figure 8.22 shows a simple network running the DSDV protocol on Ubiquest nodes. The Ubiquest simulation platform is configured using the *coloration* tab to color a route between a source node 10 and a destination node 11. The following steps are needed. First, select the corresponding relation, which is *Route*. Second, choose the appropriate provided type, which is *path*, and finally choose the color.

Using the Ubiquest simulation platform, we detect that our first implementation of DSDV had a problem because the routes were not stable. The route from the source node 10 to the destination node 11 is always flapping between three different routes as shown below:

- First route includes nodes {10, 4, 3, 8, 0, 11}
- Second route includes nodes {10, 8, 2, 5, 7, 11}
- Third route includes nodes {10, 8, 6, 5, 7, 11}

The route flapping is due to a problem in Rule (4.51). For simplicity reason, let us rewrite Rule (4.51) with the corresponding schema as shown below in Rule (8.1).

8.2 Experimentation and Validation

$$\begin{aligned}
\downarrow \text{Route}(x, \diamond y, n, s, t) : & - \text{HelloRt}(y, x, n', s), s' < s, \\
& !\text{Route}(x, y', n'', s'), n := n' + 1, \\
& t := m_time + m_timeout.
\end{aligned} \tag{8.1}$$

Schema	Description
Route(d,x,n,q,t)	Route(dest, nextHop, numberHop, sequenceNb, timeout)
HelloRt(s,d,n,q)	HelloRt(sender, destination, numberHop, sequenceNb)

Table 8.1: Schemas of the *DSDV-like* routing protocol

The flapping problem is due to the fact that Rule (8.1) updates the relation *Route* when a new route is received with a higher sequence number. Each time the source node 10 receives a fact *HelloRt*, Rule 8.1 is applied and the relation *Route* is updated because the sequence number *s* of received route is higher then the sequence number *s'* of existing route, whatever is the nexthop. This problem can be fixed by simply adding additional condition on the nexthop *y* of a route, which needs to be the same as the received one, as shown in Rule (8.2).

$$\begin{aligned}
\downarrow \text{Route}(x, \diamond y, n, s, t) : & - \text{HelloRt}(y, x, n', s), s' < s, \\
& !\text{Route}(x, y, n'', s'), n := n' + 1, \\
& t := m_time + m_timeout.
\end{aligned} \tag{8.2}$$

This problem demonstrates the utility to monitor the behavior of programs, as well as the facility of modifying the code source of declarative programs.

Tree Construction

We consider the construction of a simple spanning tree. The tree program, Rules (4.5 - 4.7), seen in Chapter 4 can be experimentally checked with the Ubiquet simulation platform. The program is creating a tree with node 0 as root. The *coloration* tab is configured to display the global tree of the root node 0 by selecting the corresponding relation *ST*, and the provided type *global*. When the simulation starts, we can immediately visualize the different steps of the creation of the tree and if there is any error. The final state of the execution of this program is shown in Figure 8.23.

Mobile Client

The following program, Rules (8.3 - 8.11), allows a client to maintain periodically a route to a server. This program is divided in three sub-programs: server – Rule (8.3), relay – Rules (8.4 - 8.7), and client – Rules (8.8 - 8.11). Servers provide services, relays maintain routes to the nearest server, and clients require shortest route to a server. This program runs on a set of Ubiquet nodes as shown in Figure 8.24.

$$\uparrow \text{RelayRep}(self, @x, self, 1) : -!\text{RelayReq}(x). \tag{8.3}$$

8.2 Experimentation and Validation

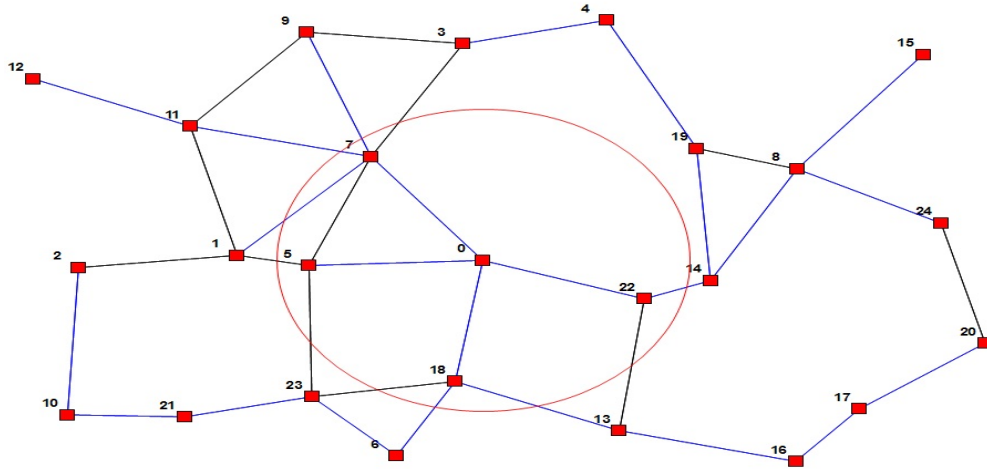


Figure 8.23: A network with Tree protocol

Schema	Description
RelayReq(r)	RelayReq(relayId)
RelayRep(s,r,x,n)	RelayRep(serverId, relayId, nodeId, numberOfHops)
Provider(s,y,n,t)	Provider(serverId, nextHop, numberOfHops, timeout)
ClientReq(x,y)	ClientReq(clientId, nodeId)
ClientRep(s,x,y,n)	ClientRep(serverId, nodeId, clientId, numberOfHops)

Table 8.2: Schemas of the *Mobile Client* protocol

$$\begin{aligned} \uparrow \text{RelayReq}(\text{self}) : & \neg \text{TimeEvent}('relay'); \\ & \neg \text{Provider}(_, _, _, _). \end{aligned} \quad (8.4)$$

$$\begin{aligned} \uparrow \text{RelayRep}(x, @y, \text{self}, n) : & \neg \text{RelayReq}(y); \\ & \text{Provider}(x, _, n', _); n := n' + 1. \end{aligned} \quad (8.5)$$

$$\begin{aligned} \downarrow \text{Provider}(x, y, n, t) : & \neg \text{RelayRep}(x, \text{self}, y, n); \\ & \neg \text{Provider}(_, _, _, _); t := \text{time} + 9. \end{aligned} \quad (8.6)$$

$$\begin{aligned} \uparrow \text{ClientRep}(x, \text{self}, @y, n) : & \neg \text{ClientReq}(y, _); \\ & \text{Provider}(x, _, n', _); n := n' + 1. \end{aligned} \quad (8.7)$$

$$\begin{aligned} \uparrow \text{ClientReq}(\text{self}, \text{self}) : & \neg \text{TimeEvent}('route'); \\ & \neg \text{Provider}(_, _, _, _). \end{aligned} \quad (8.8)$$

$$\begin{aligned} \uparrow \text{ClientReq}(\text{self}, @x) : & \neg \text{TimeEvent}('route'); \\ & \text{Provider}(_, x, _, _). \end{aligned} \quad (8.9)$$

$$\begin{aligned} \downarrow \text{Provider}(x, y, n, t) : & \neg \text{ClientRep}(x, y, \text{self}, n); \\ & \neg \text{Provider}(_, _, _, _); t := \text{time} + 9. \end{aligned} \quad (8.10)$$

$$\begin{aligned} \downarrow \text{Provider}(x, y, n, t) : & \neg \text{ClientRep}(x, y, \text{self}, n); \\ & \text{Provider}(_, x, n, _); t := \text{time} + 9. \end{aligned} \quad (8.11)$$

8.2 Experimentation and Validation

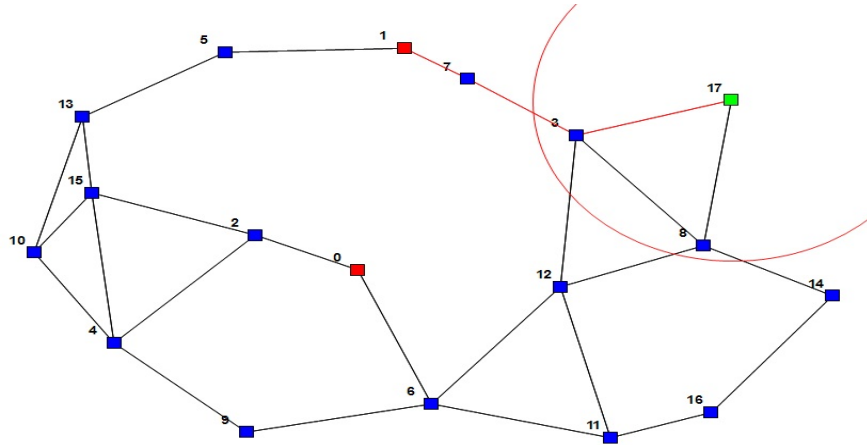


Figure 8.24: A network with Mobile Client at position p_1

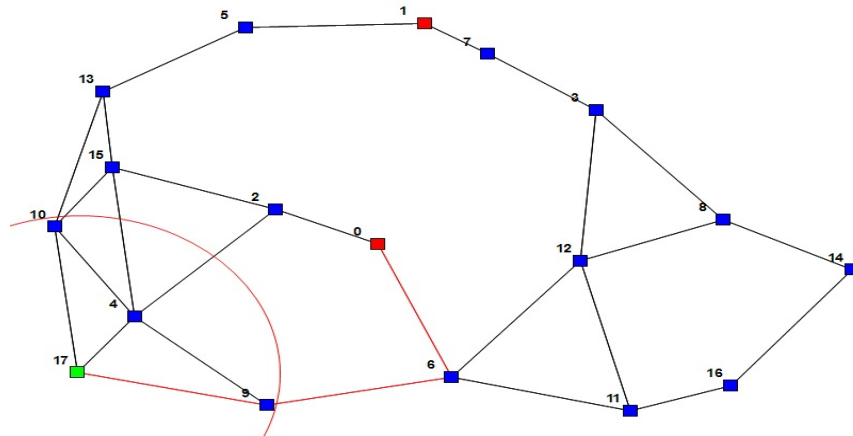


Figure 8.25: A network with Mobile Client at position p_2

The simulated network is composed of 2 servers (nodes 0 and 1), 15 relays (nodes 2 – 16) and one moving client (node 17). Using the related program, the client maintains dynamically a route to the nearest server through relays.

We are interested to visualize the route between the client and the nearest server. With the tab *coloration* of the Ubiquest simulation platform, we select the relation *provider*, the type *recursive*, and specify the color of the route. Afterwards, the Ubiquest simulation platform displays the route from the client to the nearest server as shown in Figure 8.24. When the client is moving, we notice that the route is updated on the fly to reflect new values of the data as shown in Figure 8.25.

On-demand Routing

In Chapter 5, we presented the implementation of the on-demand routing protocol, Rules (5.8 - 5.10). Figure 8.26 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.

We are interested to visualize on the fly all discovered routes between the source node and the destination. We configured the *coloration* tab, we select the relation *route*, the type *path*, and specify

8.2 Experimentation and Validation

the color of the route. After that we run the protocol.

The source node sends subqueries to its neighbors which in turn repeat the same process 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 *on-demand routing* program follows the same paths of subqueries propagation. Suppose 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. In Figure 8.26 for instance, node 5 is the first node that answers the query.

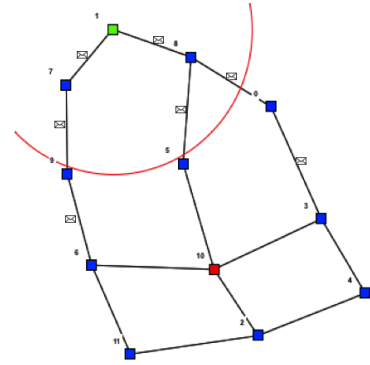


Figure 8.26: Propagation of queries

dest	nexthop	numberofhop
0010	0008	4
0010	0008	3
0010	0007	4

Figure 8.27: Visualization of ItemSet route

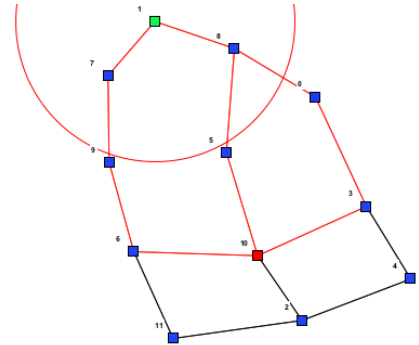


Figure 8.28: Routes coloration

Intermediate nodes aggregate answers to the source of the query. When receiving the answers, the source node 1 stores its discovered routes in the routing table as seen in Figure 8.27. Each time a route is built, it will be colored as shown in Figure 8.28. That allows us to visualize the behavior of declarative network protocols upon link or node failure or departure through direct interaction with the network.

Conclusion

We presented the Ubiquet approach which proposes a unified view for network management and data management. We described the architecture of the Ubiquet system which integrates the Netquest system for network management, with SQL-like query language, DLAQL, for data management. An important characteristic of the Ubiquet system is the execution of DLAQL queries including networking programs (e.g. routing, distributed algorithms) as optimization decisions. We presented the Ubiquet simulation platform that extends the QuestMonitor visualization tool in order to monitor the behavior of declarative programs as well as the cost of executing DLAQL queries. A series of declarative programs has been implemented and visualized over the Ubiquet simulation platform. We demonstrated the utility of the Ubiquet simulation platform which allows to detect flaws in programs, and showed the facility offered by this platform for rapid prototyping.

Conclusion

Distributed programming has tremendously gained in importance in recent years, especially with the wide development of networks, which become ubiquitous and support applications in many domains. Distributed programming though is still a very complex task. The objective of this thesis is to facilitate distributed programming, and to obtain distributed algorithms more likely to be correct, verifiable, extensible, with a high level of resilience and persistence.

To achieve our objective, we first studied the difficulties and challenges of designing and implementing distributed systems and applications. As a result, we found that providing high-level abstraction for their communication model, in particular for their target destinations, offers a great potential to facilitate their programming and to satisfy some of their properties such as persistence of data as well as resilience of systems. We proposed solutions based on communication with intensional destinations as well as on declarative programming. The latter provides a new level of abstraction that not only simplifies programming but also satisfies some properties such as extensibility and concurrency.

Summary of Contributions

The main contributions of this thesis can be summarized as follows:

- We developed a framework that defines a new level of abstraction for the destination of messages whose destination is specified both extensionally, by an explicit address (e.g. IP, Id, MAC, etc.), and intensionally, by an implicit address specified by a selection criteria declared upon application specification. The selection criteria is a set of properties which can be for instance a very simple property based on local data of a node (e.g. local SQL-like query), or it can be more complex expressed by a distributed program. This framework permits to program (distributed) applications in a message-oriented manner, allowing messages with extensional/intensional destination, that are solved in the network while they are traveling. If the node associated with the extensional destination is unreachable, the intensional destination is evaluated on the fly, and a new node is identified as (extensional) destination of the message. This ensures persistence of data in messages, as well as resilience of the system supporting the applications. We showed that this framework simplifies the way of expressing a variety of distributed programs [12, 14], as well as provides resilience and persistence for applications running over WSNs and overlay networks [6, 11, 14].
- To handle efficiently the intensional destination selection criteria, we formally defined the data-centric language, Questlog, that specifies the intensional destinations as queries, and programs complex strategies to evaluate them. The Questlog language has its roots in logic programming and recursive query languages [136]. It is based on the observation that recursive queries offer a

Conclusion

natural way to express reactive network programs that themselves exhibit recursive properties. Questlog follows the *pull* mode, in contrast to the traditional recursive centralized language Datalog which follows the *push* mode, and enables distributed computations.

We showed that the Questlog language is compact and natural to express aggregation, a variety of reactive protocols such as the on-demand routing as well as reactive application queries, often resulting in orders of magnitude savings in code size. Questlog programs are compiled into SQL queries which are then executed using a distributed engine to implement the programs. 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.

- To facilitate programming distributed systems and to validate our framework, we developed a setting that allows, under some restrictions, to distribute seamlessly client/server applications into P2P systems using declarative overlays. We showed that networking protocols could be written as simple, very concise programs consisting of a few dozen of rules. We described the overlay which is defined by a combination of an ad hoc routing protocol, DSDV, together with a DHT such as Chord. We then showed that applications coded as queries in a client/server framework could be ported seamlessly, that is without modifying the initial queries, to the distributed environment. The distributed system based on the DHT ensures the tasks of the centralized server in a fully distributed manner, by relying in the nodes which handle horizontal fragments of the relations, and communicate with other nodes to solve queries. The communication between nodes relies on extensional/intensional destination, which can be evaluated on the fly to ensure the persistence of data and queries. We considered the promising example of multiplayer online games, which can be fully described in a data centric fashion, and showed how it can be seamlessly distributed. We made experiments on the QuestMonitor platform to demonstrate the robustness of the approach.
- To demonstrate our framework in the domain of WSNs, which have achieved considerable success in recent years, we proposed to program WSN applications using messages with intensional destinations, which are delivered to interested sensor nodes that satisfy certain properties. We showed that providing high-level abstraction for destinations yields modularity. We presented programs expressed in Questlog to reactively collect and aggregate data from a subset of powerful nodes in a network. We proposed as well a dynamic clustering protocol that allows to construct clusters to aggregate efficiently sensors collected data. We specified intensionally the cluster heads which are evaluated on the fly when messages are traveling. We showed that the protocol adapts dynamically to topology changes, and provides high load balancing. The mobility of the code that distinguishes the intensional destination selection criteria offers an elegant formulation and dynamic modification, which facilitates programming. Thanks to the intensional destination which allows to (i) program applications in a simple, flexible, and modular manner, and (ii) guarantee an increased persistence for data traveling in the network as well as resilience for the system.

Perspectives

The topics we have presented in this thesis open several research perspectives.

Programming strategies of intensional destinations

We presented in Chapter 3 a framework that allows to program distributed algorithms in a message-oriented manner. This framework offers a new model of messages with destination specified both extensionally, by an address, and intensionally, by some selection criteria. Consider a selection criteria represented by a query, then only nodes that satisfy the intensional destination query are allowed to compute the message's content. Another strategy can be used as well. The intensional destination query can be moved to the payload of the message. Then the message is diffused in the network, and the message's content will be evaluated by all nodes. This mechanism allows to balance the load between the payload and the destination, leading to different evaluation schemes. We plan to further study the different programming strategies offered by intensional destinations, and make experiments to evaluate the performance of each strategy using different types of networks (e.g. Ad hoc, WSN) with different topologies. We plan to implement the different strategies and let the system choose the appropriate one. The main question is how to specify in a dynamic manner the strategy to be used? We believe that the decision can sometimes be made based on local properties such as energy, memory, etc., and potentially other nodes properties.

Experimentation

The Questlog language introduced in Chapter 5 is well adapted to reactive protocols as well as to complex application queries. We used the QuestMonitor platform, which is a visualization tool, to interact with a network and visualize the behavior of programs. We ran some Questlog programs on the visualization tool, but we did not study their performance. We need a real simulator such as WSNNet or ns3 to realize such task. The related Questlog system (Netquest) can be implemented as a module on the simulator. Due to lack of time, we did not implement this module, which we plan to build it. This allows to realize experimentations and study the performance of Questlog programs. These experimentations will allow to measure the impact of routing with intensional destinations on the delay, the complexity in terms of communication messages, the execution time of Questlog queries, the latency, as well as the overhead of programs. They will also allow to investigate potential optimization that can be made on the Netquest system, on Questlog programs, as well as on selecting intensional destination best strategy via implicit acquisition of information.

Likewise, we presented in Chapter 7 a novel clustering protocol, DICE, that decomposes the network into a set of dynamic clusters to aggregate efficiently sensors collected data. To evaluate the protocol, we started the simulations on the WSNNet [162] network simulator. Our first results are promising but they are not enough mature to be included in this dissertation. We plan to make a deep study on the performance of the protocol and a comparison between DICE and other classical clustering approaches.

Declarative programming environment

In this dissertation, we used two declarative languages Netlog (*push* mode), and Questlog (*pull* mode) which has been introduced in Chapter 5. We noticed during the programming of distributed algorithms that, for some protocols such as Chord [148], the interaction between Questlog and

Conclusion

Netlog is essential for more efficient implementation. Ideally, when programming in declarative languages, programmers usually do not need to specify *how* computation is done, but rather *what* is to be computed. This is in contrast to imperative programming, which requires a detailed description of the algorithm of computation. In the literature, different languages have been proposed with different levels of declarativity [107]. It will be interesting to introduce a new programming environment that radically simplifies wide range of tasks, and explores the declarative approach to the full potential. This should be done taking into consideration a satisfying high level of declarativity. In particular, programmers need to specify the desired results, and not necessary the algorithms to compute them, leaving to systems the process of generating the appropriate (distributed) algorithms (either in the *pull* or in the *push* mode, or both).

Generalize distribution environment

In Chapter 4, we developed an environment that allows, under some restrictions, to distribute seamlessly client/server applications into P2P systems. In this environment, we specified *a priori* some policies (restrictions) to be respected by any application that will be implemented. They are used to specify index placement for smooth and efficient distribution for data and queries following the unicast mode. However, the imposed restrictions on allowed queries limit the possible applications. Consider for instance an online multiplayer game, e.g. Counter Strike, in which two teams, terrorists and counter terrorists, try to eliminate each other during rounds. Each player can perform various actions such as killing opponents, buying equipments, etc. Suppose that a player fires a query that provides the position of all players of their team on the global map, and not only in the local area of the player. Global queries of all the players in the whole virtual world cannot be handled easily, and would require either relaxing the assumption of allowed queries, or replicating tremendously the data. There is a tradeoff between efficiency and generality, between allowed queries ensuring unicast and general queries, requiring broadcast in general. We propose to generalize this environment in order to allow diverse applications to be implemented, in a client/server architecture, and distributed into P2P systems, while keeping the system efficient. In particular, we plan to relax the predefined policies, and allows programmers to define their applications in a client/server architecture using SQL or SQL-like languages. The system deduces index placement *a posteriori*, based on soft-business rules specified upon writing the applications, in order to potentially enable efficient distribution of large classes of applications data and queries.

Bibliography

- [1] Martín Abadi and Boon Thau Loo. Towards a declarative language and system for secure networking. In *Proc. NETB'07*, pages 1–6. USENIX Association, 2007.
- [2] Ameer Ahmed Abbasi and Mohamed F. Younis. A survey on clustering algorithms for wireless sensor networks. *Computer Communications*, 30(14-15):2826–2841, 2007.
- [3] Serge Abiteboul, Émilien Antoine, and Julia Stoyanovich. The WebdamLog System: Managing Distributed Knowledge on the Web. In *Base de données avancées*, Clermont-Ferrand, France, October 2012.
- [4] Serge Abiteboul, Meghyn Bienvenu, Alban Galland, and Emilien Antoine. A rule-based language for web data management. In *PODS*, pages 293–304, 2011.
- [5] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silber-schatz. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.
- [6] Ahmad Ahmad-Kassem, Eric Bellemon, and Stéphane Grumbach. Seamless distribution of data centric applications through declarative overlays. *BDA'11: 27ème journées Bases de Données Avancées, Rabat, Maroc*, October 2011.
- [7] Ahmad Ahmad-Kassem, Christophe Bobineau, Christine Collet, Etienne Dublé, Stéphane Grumbach, Fuda Ma, Lourdes Martínez, and Stéphane Ubéda. A data-centric approach for networking applications. In *DATA 2012 - Proceedings of the International Conference on Data Technologies and Applications, Rome, Italy, 25-27 July, 2012*, pages 147–152, 2012.
- [8] Ahmad Ahmad-Kassem, Christophe Bobineau, Christine Collet, Etienne Dublé, Stéphane Grumbach, Fuda Ma, Lourdes Martínez, and Stéphane Ubéda. Ubiquet, for rapid prototyping of networking applications. In *16th International Database Engineering & Applications Symposium, IDEAS '12, Prague, Czech Republic, August 8-10, 2012*, pages 187–192, 2012.
- [9] Ahmad Ahmad-Kassem, Christophe Bobineau, Christine Collet, Etienne Dublé, Stéphane Grumbach, Fuda Ma, Lourdes Martínez, and Stéphane Ubéda. Ubiquet for declarative and adaptive programming of networking applications. *International Journal of Advanced Computer Science*, 3(2), 2013.
- [10] Ahmad Ahmad-Kassem, Christophe Bobineau, and Stéphane Grumbach. D3.1 Network query language design, 2010. Ubiquet Project ANR-09-BLAN-0131-01.
- [11] Ahmad Ahmad-Kassem and Stéphane Grumbach. Distribution d'applications client-serveur sur des réseaux déclaratifs. *Ingénierie des Systèmes d'Information*, 17(5):113–138, 2012.
- [12] Ahmad Ahmad-Kassem, Stéphane Grumbach, and Stéphane Ubéda. Messages with implicit destinations as mobile agents. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, AGERE! '12, Tucson, Arizona, USA*, pages 107–118, 2012.
- [13] Ahmad Ahmad-Kassem, Lourdes-Angelica Martinez-Medina, and Etienne Dublé. D5.1 Implementation of the simulation environment, 2013. Ubiquet Project ANR-09-BLAN-0131-01.
- [14] Ahmad Ahmad-Kassem, Fabrice Valois, Ibrahim Amadou, and Stéphane Grumbach. Data

BIBLIOGRAPHY

- aggregation through intensional clustering. Draft paper, March 2013.
- [15] Kemal Akkaya and Mohamed F. Younis. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks*, 3(3):325–349, 2005.
 - [16] Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. Dks (n, k, f): A family of low communication, scalable and fault-tolerant infrastructures for p2p applications. In *CCGRID*, pages 344–350, 2003.
 - [17] Gustavo Alonso, Evangelos Kranakis, Cindy Sawchuk, Roger Wattenhofer, and Peter Widmayer. Probabilistic protocols for node discovery in ad hoc multi-channel broadcast networks. In *Proc. ADHOC-NOW'03*, 2003.
 - [18] Peter Alvaro, William Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell C Sears. Dedalus: Datalog in time and space. Technical report, EECS Department, University of California, Berkeley, Dec 2009.
 - [19] Michael P. Ashley-Rollman, Michael De Rosa, Siddhartha S. Srinivasa, Padmanabhan Pillai, Seth Copen Goldstein, and Jason D. Campbell. Declarative programming for modular robots. In *Workshop on Self-Reconfigurable Robots/Systems and Applications at IROS '07*, October 2007.
 - [20] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Wiley-Interscience, 2004.
 - [21] Emmanuel Baccelli. OLSR Trees: A Simple Clustering Mechanism for OLSR. In International Federation for Information Processing (IFIP), editor, *Challenges in Ad Hoc Networking*, volume 197, pages 265–274. Springer, October 2006.
 - [22] Francois Bancilhon. Naive evaluation of recursively defined relations. In *On knowledge base management systems: integrating artificial intelligence and database technologies*, 1986.
 - [23] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey 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.
 - [24] S. Banerjee and S. Khuller. A clustering scheme for hierarchical control in multi-hop wireless networks. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 1028–1037 vol.2, 2001.
 - [25] Michel Bauderon, Stephane Grumbach, Daqing Gu, Xin Qi, Wenwu Qu, Kun Suo, and Yu Zhang. Programming imote networks made easy. In *The Fourth International Conference on Sensor Technologies and Applications*, pages 539–544. IEEE Computer Society, 2010.
 - [26] Yigal Bejerano, Yuri Breitbart, Minos N. Garofalakis, and Rajeev Rastogi. Physical topology discovery for large multi-subnet networks. In *Proc. INFOCOM'03*, 2003.
 - [27] Yigal Bejerano, Yuri Breitbart, Ariel Orda, Rajeev Rastogi, and Alexander Sprintson. Algorithms for computing qos paths with restoration. *IEEE/ACM Trans. Netw.*, 13(3), 2005.
 - [28] Nalini Moti Belaramani, Jiandan Zheng, Amol Nayate, Robert Soulé, Michael Dahlin, and Robert Grimm. Pads: A policy architecture for distributed storage systems. In *NSDI*, pages 59–74, 2009.
 - [29] Eric Bellemon, Vincent Dubosclard, Stéthane Grumbach, and Kun 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.
 - [30] BitTorrent. <http://www.bittorrent.com/>.

BIBLIOGRAPHY

- [31] Boris Jan Bonfils and Philippe Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *IPSN*, pages 47–62, 2003.
- [32] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Mobile Data Management*, pages 3–14, 2001.
- [33] Sonja Buchegger, Doris Schiöberg, Le-Hung Vu, and Anwitaman Datta. Peer-to-peer social networking: early experiences and insights. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems*, 2009.
- [34] Matthew Caesar, Miguel Castro, Edmund B. Nightingale, Greg O’Shea, and Antony Rowstron. Virtual ring routing: network routing inspired by dhts. *SIGCOMM Comput. Commun. Rev.*, 36:351–362, 2006.
- [35] Antonio Carzaniga and Cyrus P. Hall. Content-based communication: a research agenda. In *SEM*, pages 2–8, 2006.
- [36] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Content-based addressing and routing: A general model and its application. Technical Report CU-CS-902-00, Department of Computer Science, University of Colorado, January 2000.
- [37] Antonio Carzaniga, Matthew J. Rutherford, and Alexander L. Wolf. A routing scheme for content-based networking. In *INFOCOM*, 2004.
- [38] Antonio Carzaniga and Alexander L. Wolf. Content-based networking: A new communication infrastructure. In *Infrastructure for Mobile and Wireless Systems*, pages 59–68, 2001.
- [39] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *SIGCOMM*, pages 163–174, 2003.
- [40] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure routing for structured peer-to-peer overlay networks. *SIGOPS Oper. Syst. Rev.*, 36, December 2002.
- [41] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.
- [42] Haowen Chan and Adrian Perrig. Ace: An emergent algorithm for highly uniform cluster formation. In *EWSN*, pages 154–171, 2004.
- [43] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In *SIGCOMM*, pages 407–418, 2003.
- [44] Dhruv Chopra, Henning Schulzrinne, Enrico Marocco, and Emil Ivov. Peer-to-peer overlays for real-time communication: Security issues and solutions. *IEEE Communications Surveys and Tutorials*, 11(1), 2009.
- [45] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph M. Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The design and implementation of a declarative sensor network system. In *Sensys*, pages 175–188, 2007.
- [46] Maurice Chu, Horst W. Haussecker, and Feng Zhao. Scalable information-driven sensor querying and routing for ad hoc heterogeneous sensor networks. *IJHPCA*, 16(3):293–313, 2002.
- [47] T. Clausen and P. Jacquet. Optimized Link State Routing Protocol (OLSR). RFC 3626 (Experimental), October 2003.
- [48] Bram Cohen. Incentives build robustness in bittorrent, 2003.
- [49] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed systems - concepts and designs (5. ed.)*. International computer science series. Addison-Wesley-

BIBLIOGRAPHY

- Longman, 2012.
- [50] Gianpaolo Cugola, Alessandro Margara, and Matteo Migliavacca. Context-aware publish-subscribe: Model, implementation, and evaluation. In *ISCC*, pages 875–881, 2009.
 - [51] Gianpaolo Cugola and Matteo Migliavacca. A context and content-based routing protocol for mobile sensor networks. In *EWSN*, pages 69–85, 2009.
 - [52] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
 - [53] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
 - [54] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.
 - [55] Alan J. Demers, Johannes Gehrke, Rajmohan Rajaraman, Agathoniki Trigoni, and Yong Yao. The cougar project: a work-in-progress report. *SIGMOD Record*, 32(4):53–59, 2003.
 - [56] Yuxin Deng, Stéphane Grumbach, and Jean-François 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.
 - [57] Tony Ducrocq, Nathalie Mitton, and Michaël Hauspie. Clustering pour l’optimisation de la durée de vie des réseaux de capteurs sans fil. In Nicolas Mathieu, Fabien et Hanusse, editor, *14èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (AlgoTel)*, La Grande Motte, France, 2012.
 - [58] Jason Eisner, Eric Goldlust, and Noah A. Smith. Dyna: A declarative language for implementing dynamic programs. In *In Proc. of ACL*, page 2004, 2004.
 - [59] Deborah Estrin, Ramesh Govindan, John S. Heidemann, and Satish Kumar. Next century challenges: Scalable coordination in sensor networks. In *MOBICOM*, pages 263–270, 1999.
 - [60] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.
 - [61] Patrick Th. Eugster, Benoit Garbinato, and Adrian Holzer. Location-based publish/subscribe. In *NCA*, pages 279–282, 2005.
 - [62] Ronaldo A. Ferreira, Murali Krishna Ramanathan, Asad Awan, Ananth Grama, and Suresh Jagannathan. Search with probabilistic guarantees in unstructured peer-to-peer networks. In *Peer-to-Peer Computing*, pages 165–172, 2005.
 - [63] Gnutella Development Forum. The gnutella v0.6 protocol.
 - [64] Deepak Ganesan, Ramesh Govindan, Scott Shenker, and Deborah Estrin. Highly-resilient, energy-efficient multipath routing in wireless sensor networks. *Mobile Computing and Communications Review*, 5(4):11–25, 2001.
 - [65] Gnucleus. The gnutella web caching system. <http://www.gnucleus.org/gwebcache/>.
 - [66] Stéphane Grumbach and Fang Wang. Netlog, a rule-based language for distributed programming. In *PADL’10, Twelfth International Symposium on Practical Aspects of Declarative Languages*, Madrid, Spain, 2010.
 - [67] Stéphane Grumbach and Fang Wang. Netlog, a rule-based language for distributed programming. In *Proc. PADL’10*, volume 5937 of *LNCS*, pages 88–103, 2010.
 - [68] Thorsten Hampel, Thomas Bopp, and Robert Hinn. A peer-to-peer architecture for massive multiplayer online games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, 2006.

BIBLIOGRAPHY

- [69] Wendi B. Heinzelman, Anantha P. Chandrakasan, and Hari Balakrishnan. An application-specific protocol architecture for wireless microsensor networks. *IEEE Transactions on Wireless Communications*, 1(4):660–670, 2002.
- [70] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *HICSS*, 2000.
- [71] Joseph M. Hellerstein. The declarative imperative, experience and conjecture in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.
- [72] Adrian Holzer, Lukasz Ziarek, K. R. Jayaram, and Patrick Eugster. Putting events in context: aspects for event-based distributed programming. In *AOSD*, pages 241–252, 2011.
- [73] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MOBICOM*, pages 56–67, 2000.
- [74] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John S. Heidemann, and Fabio Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw.*, 11(1):2–16, 2003.
- [75] Matteo Interlandi, Letizia Tanca, and Sonia Bergamaschi. Datalog in time and space, synchronously. In *AMW 2013: Proceedings of the 7th Alberto Mendelzon International Workshop on Foundations of Data Management*, Puebla/Cholula, Mexico, 2013.
- [76] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca Braynard. Networking named content. In *CoNEXT*, pages 1–12, 2009.
- [77] Trevor Jim. Sd3: A trust management system with certified evaluation. In *IEEE Symposium on Security and Privacy*, pages 106–115, 2001.
- [78] Trevor Jim and Dan Suciu. Dynamically distributed query evaluation. In *PODS*, 2001.
- [79] James Jobin, Zhenqiang Ye, Honomount Rawat, and Srikanth V. Krishnamurthy. A lightweight framework for source-to-sink data transfer in wireless sensor networks. In *BROAD-NETS*, pages 756–766, 2005.
- [80] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, 1997.
- [81] Wang Ke, Salma Abu Ayyash, Thomas D. C. Little, and Prithwish Basu. Attribute-based clustering for information dissemination in wireless sensor networks. In *SECON*, pages 498–509, 2005.
- [82] Bjorn Knutsson, Massively Multiplayer Games, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM*, 2004.
- [83] Balachander Krishnamurthy and Jia Wang. On network-aware clustering of web clients. In *SIGCOMM*, pages 97–110, 2000.
- [84] Balachander Krishnamurthy, Jia Wang, and Yinglian Xie. Early measurements of a cluster-based architecture for p2p systems. In *Internet Measurement Workshop*, pages 105–109, 2001.
- [85] Lars Kulik, Egemen Tanin, and Muhammad Umer. Efficient data collection and selective queries in sensor networks. In *GSN*, pages 25–44, 2006.
- [86] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *PODS*, pages 1–12, 2005.

BIBLIOGRAPHY

- [87] Christof Leng, Wesley W. Terpstra, Bettina Kemme, Wilhelm Stannat, and Alejandro P. Buchmann. Maintaining replicas in unstructured p2p systems. In *CoNEXT*, page 19, 2008.
- [88] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *PADL*, pages 58–73, 2003.
- [89] David Liben-Nowell, Hari Balakrishnan, and David R. Karger. Analysis of the evolution of peer-to-peer systems. In *PODC*, pages 233–242, 2002.
- [90] Changbin Liu, Ricardo Correa, Xiaozhou Li, Prithwish Basu, Boon Thau Loo, and Yun Mao. Declarative policy-based adaptive manet routing. In *ICNP*, pages 354–363, 2009.
- [91] Changbin Liu, Yun Mao, Mihai Oprea, Prithwish Basu, and Boon Thau Loo. A declarative perspective on adaptive manet routing. In *Proc. PRESTO '08*, pages 63–68. ACM, 2008.
- [92] J. Liu, S. G. Rao, B. Li, and H. Zhang. Opportunities and challenges of peer-to-peer internet video broadcast. *Special Issue on Recent Advances in Distributed Multimedia Communications, Vol. 96, No. 1, pp. 11-24*, 2008.
- [93] Boon Thau Loo. *The Design and Implementation of Declarative Networks*. PhD thesis, University of California, Berkeley, 2006.
- [94] Boon Thau Loo, Tyson Condie, Minos N. Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: language, execution and optimization. In *Proc. ACM SIGMOD'06*, 2006.
- [95] Boon Thau Loo, Tyson Condie, Minos N. Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, 2009.
- [96] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *Proc. SOSP'05*, 2005.
- [97] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. *SIGOPS Oper. Syst. Rev.*, 39:75–90, 2005.
- [98] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *Proc. ACM SIGCOMM '05*, 2005.
- [99] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 7(1-4):72–93, 2005.
- [100] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.
- [101] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30, 2005.
- [102] Amirhossein Malekpour, Antonio Carzaniga, Fernando Pedone, and Giovanni Toffetti Carughi. End-to-end reliability for best-effort content-based publish/subscribe networks. In *DEBS*, pages 207–218, 2011.
- [103] G. Malkin. RFC 2453: RIP Version 2 . Technical report, IETF, 1998.
- [104] Ching man Au Yeung, Ilaria Liccardi, Kanghao Lu, Oshani Seneviratne, and Tim Berners-Lee. Decentralization: The future of online social networking. In *W3C Workshop on the Future of Social Networking Position Papers*, 2009.
- [105] Arati Manjeshwar and Dharma P. Agrawal. Teen: A routing protocol for enhanced efficiency in wireless sensor networks. In *IPDPS*, page 189, 2001.

BIBLIOGRAPHY

- [106] Arati Manjeshwar and Dharma P. Agrawal. Apteen: A hybrid protocol for efficient routing and comprehensive information retrieval in wireless sensor networks. In *IPDPS*, 2002.
- [107] Yun Mao. On the declarativity of declarative networking. *Operating Systems Review*, 43(4):19–24, 2009.
- [108] Yun Mao, Boon Thau Loo, Zachary G. Ives, and Jonathan M. Smith. Mosaic: unified declarative platform for dynamic overlay composition. In *CoNEXT*, page 5, 2008.
- [109] Pedro Jose Marron and Daniel Minder. *Embedded WiSeNts Research Roadmap*. Embedded WiSeNts Consortium, 2006.
- [110] Lourdes Martínez, Christine Collet, Christophe Bobineau, and Etienne Dublé. The qol approach for optimizing distributed queries without complete knowledge. In *IDEAS*, pages 91–99, 2012.
- [111] J. Legatheaux Martins and S. Duarte. Routing algorithms for content-based publish/subscribe systems. *IEEE Communications Surveys and Tutorials*, 01 2010.
- [112] Tova Milo, Tal Zur, and Elad Verbin. Boosting topic-based publish-subscribe systems with dynamic clustering. In *SIGMOD Conference*, pages 749–760, 2007.
- [113] Dejan S. Milojevic and Fred B. Schneider. Interview - Fred B. Schneider on distributed computing. *IEEE Distributed Systems Online*, 1(1), 2000.
- [114] Nathalie Mitton, Anthony Busson, and Eric Fleury. Self-organization in large scale ad hoc networks. In *Mediterranean ad hoc Networking Workshop (MedHocNet'04)*, page 0000, Bodrum, Turquie, June 2004.
- [115] J. Moy. RFC 2328: OSPF Version 2. Technical report, IETF, 1998.
- [116] Krishna Nadiminti, Marcos Dias De Assuncao, and Rajkumar Buyya. Distributed systems and recent innovations: Challenges and benefits, 2006.
- [117] Napster. <http://www.napster.com/>.
- [118] Juan A. Navarro and Andrey Rybalchenko. Operational semantics for declarative networking. In *Proc. PADL '09*, pages 76–90. Springer, 2009.
- [119] B. Clifford Neuman. Scale in distributed systems. In *Readings in Distributed Computing Systems*, pages 463–489. IEEE Computer Society Press, 1994.
- [120] Navid Nikaein and Christian Bonnet. Topology management for improving routing and network performances in mobile ad hoc networks. *MONET*, 9(6):583–594, 2004.
- [121] Navid Nikaein, Houda Labiod, and Christian Bonnet. Ddr: distributed dynamic routing algorithm for mobile ad hoc networks. In *MobiHoc*, pages 19–27, 2000.
- [122] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.
- [123] OMG. Common Object Request Broker Architecture (CORBA/IIOP).v3.1. Technical report, OMG, January 2008.
- [124] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD Conference*, pages 165–178, 2009.
- [125] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental), July 2003.
- [126] Charles Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsv) for mobile computers. In *ACM SIGCOMM'94 Conference on Communications*

BIBLIOGRAPHY

- Architectures, Protocols and Applications*, pages 234–244, 1994.
- [127] Charles 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.
 - [128] Charles E. Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. In *ACM Conference on Communications Architectures, Protocols and Applications, SIGCOMM '94, London, UK*, pages 234–244. ACM, ACM, August 1994.
 - [129] Per Persson. Exms: an animated and avatar-based messaging system for expressive peer communication. In *Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*, GROUP '03, 2003.
 - [130] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *SPAA*, pages 311–320, 1997.
 - [131] Gregory J. Pottie and William J. Kaiser. Wireless integrated network sensors. *Commun. ACM*, 43(5):51–58, 2000.
 - [132] Hadoop Project. <http://hadoop.apache.org/>.
 - [133] Hive Project. <http://hive.apache.org/>.
 - [134] Ubiquist Project. <http://ubiquist.imag.fr/>.
 - [135] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2003.
 - [136] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23:125–149, 1993.
 - [137] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of deductive database systems. *J. Log. Program.*, 23(2):125–149, 1995.
 - [138] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.
 - [139] Y. Rekhter, T. Li, and S. Hares. RFC 4271: A Border Gateway Protocol 4 (BGP-4). Technical report, IETF, 2006.
 - [140] J. Reynolds and S. Ginoza. Internet Official Protocol Standards. RFC 3700 (Historic), July 2004. Obsoleted by RFC 5000.
 - [141] Sean Rhea, Byung-Gon Chun, John Kubiawicz, and Scott Shenker. Fixing the embarrassing slowness of opendht on planetlab. In *Proceedings of the 2nd conference on Real, Large Distributed Systems - Volume 2, WORLDS'05*, pages 25–30, Berkeley, CA, USA, 2005. USENIX Association.
 - [142] Sean C. Rhea, Brighten Godfrey, Brad Karp, John Kubiawicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. Opendht: a public dht service and its uses. In *SIGCOMM*, pages 73–84, 2005.
 - [143] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.
 - [144] Mahadev Satyanarayanan. On the influence of scale in a distributed system. In *ICSE*, pages 10–18, 1988.
 - [145] Thirunavukkarasu Sivaharan, Gordon S. Blair, and Geoff Coulson. Green: A configurable and re-configurable publish-subscribe middleware for pervasive computing. In *OTM Conferences (1)*, pages 732–749, 2005.
 - [146] Stanislava Soro and Wendi B. Heinzelman. Cluster head election techniques for coverage

BIBLIOGRAPHY

- preservation in wireless sensor networks. *Ad Hoc Networks*, 7(5):955–972, 2009.
- [147] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom. Operator placement for in-network stream query processing. In *Proc. POCS'05*, pages 250–258, 2005.
- [148] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31, 2001.
- [149] Netquest System. Netlog protocols. https://gforge.inria.fr/docman/?group_id=1192&view=listfile&dirid=2165.
- [150] Cisco Systems. Enhanced interior gateway routing protocol, 2005. Document ID 16406.
- [151] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms* (2. ed.). Pearson Education, 2007.
- [152] Wesley W. Terpstra, Jussi Kangasharju, Christof Leng, and Alejandro P. Buchmann. Bubblestorm: resilient, probabilistic, and exhaustive peer-to-peer search. In *SIGCOMM*, pages 49–60, 2007.
- [153] The Gnutella Protocol Specification v0.4.
- [154] Laurent Vieille. Recursive axioms in deductive databases: The query/subquery approach. In *Expert Database Conf.*, pages 253–267, 1986.
- [155] N. Vljajic and D. Xia. Wireless sensor networks: To cluster or not to cluster? In *Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks, WOWMOM '06*, pages 258–268, Washington, DC, USA, 2006. IEEE Computer Society.
- [156] David W. Wall. Messages as active agents. In *POPL*, pages 34–39, 1982.
- [157] Anduo Wang, Prithwish Basu, Boon Thau Loo, and Oleg Sokolsky. Declarative network verification. In *Proc. PADL '09*, pages 61–75. Springer, 2009.
- [158] Tianqi Wang, Wendi Rabiner Heinzelman, and Alireza Seyedi. Maximization of data gathering in clustered wireless sensor networks. In *GLOBECOM*, pages 1–5, 2010.
- [159] Mark Ward. How the web went world wide, 2006. <http://news.bbc.co.uk/2/hi/science/nature/5242252.stm>.
- [160] Tom White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale* (3. ed., revised and updated). O'Reilly, 2012.
- [161] Walker M. White, Alan J. Demers, Christoph Koch, Johannes Gehrke, and Rajmohan Rajagopalan. Scaling games to epic proportion. In *SIGMOD Conference*, pages 31–42, 2007.
- [162] WSNet. An event-driven simulator for large scale wireless networks. <http://wsnet.gforge.inria.fr/>.
- [163] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3):9–18, 2002.
- [164] Yong Yao and Johannes Gehrke. Query processing in sensor networks. In *CIDR*, 2003.
- [165] Fan Ye, Gary Zhong, Songwu Lu, and Lixia Zhang. Gradient broadcast: A robust data delivery protocol for large scale sensor networks. *Wireless Networks*, 11(3):285–298, 2005.
- [166] Hongwei Zhang and Anish Arora. Gs³: scalable self-configuration and self-healing in wireless sensor networks. *Computer Networks*, 43(4):459–480, 2003.
- [167] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

BIBLIOGRAPHY

- [168] Wenchao Zhou, Yun Mao, Boon Thau Loo, and Martín Abadi. Unified declarative platform for secure networked information systems. In *ICDE*, pages 150–161, 2009.
- [169] Wenchao Zhou, Micah Sherr, William R. Marczak, Zhuoyao Zhang, Tao Tao, Boon Thau Loo, and Insup Lee. Towards a data-centric view of cloud security. In *Second International CIKM Workshop on Cloud Data Management, CloudDB*, pages 25–32, 2010.