Università degli Studi di Bologna

FACOLTA' DI INGEGNERIA

DOTTORATO DI RICERCA IN INGEGNERIA ELETTRONICA,
INFORMATICA E DELLE TELECOMUNICAZIONI
Ciclo XX

Management and routing algorithms for ad-hoc and sensor networks

Presentata da
Dott. Gabriele Monti

Coordinatore Dottorato
Chiar.mo Prof. Ing. Paolo Bassi

Relatore
Chiar.mo Prof. Ing. Claudio Sartori

Settore scientifico disciplinare di afferenza
ING-INF/05
Esame finale anno 2008

Università degli Studi di Bologna

FACOLTA' DI INGEGNERIA

DOTTORATO DI RICERCA IN INGEGNERIA ELETTRONICA,
INFORMATICA E DELLE TELECOMUNICAZIONI
Ciclo XX

Management and routing algorithms for ad-hoc and sensor networks

Presentata da
Dott. Gabriele Monti

....

Coordinatore Dottorato
Chiar.mo Prof. Ing. Paolo Bassi

....

Relatore
Chiar.mo Prof. Ing. Claudio Sartori

....

Settore scientifico disciplinare di afferenza
ING-INF/05
Esame finale anno 2008

Abstract

Large scale wireless ad-hoc networks of computers, sensors, PDAs etc. (i.e. nodes) are revolutionizing connectivity and leading to a paradigm shift from centralized systems to highly distributed and dynamic environments. An example of ad-hoc networks are sensor networks, which are usually composed by small units able to sense and transmit to a sink elementary data which are successively processed by an external machine. Recent improvements in the memory and computational power of sensors, together with the reduction of energy consumptions, are rapidly changing the potential of such systems, moving the attention towards data-centric sensor networks. A plethora of routing and data management algorithms have been proposed for the network path discovery ranging from broadcasting/flooding-based approaches to those using global positioning systems (GPS).

We studied W-Grid, a novel decentralized infrastructure that organizes wireless devices in an ad-hoc manner, where each node has one or more virtual coordinates through which both message routing and data management occur without reliance on either flooding/broadcasting operations or GPS. The resulting ad-hoc network does not suffer from the dead-end problem, which happens in geographic-based routing when a node is unable to locate a neighbor closer to the destination than itself.

W-Grid allow multi-dimensional data management capability since nodes' virtual coordinates can act as a distributed database without needing neither special implementation or reorganization. Any kind of data (both single and multi-dimensional) can be distributed, stored and managed. We will show how a location service can be easily implemented so that any search is reduced to a simple query, like for any other data type.

W-Grid has then been extended by adopting a replication methodology. We called the resulting algorithm W^R-Grid. Just like W-Grid, W^R-Grid acts as a distributed database without needing neither special implementation nor reorganization and any kind of data can be distributed, stored and managed. We have evaluated the benefits of replication on data management, finding out, from experimental results, that it can halve the average number of hops in the network. The direct consequence of this fact are a significant improvement on energy consumption and a workload balancing among sensors (number of messages routed by each node). Finally, thanks to the replications, whose number can be arbitrarily chosen, the resulting sensor network can face sensors disconnections/connections, due to failures of sensors, without data loss.

Another extension to W-Grid is W*-Grid which extends it by strongly improving network recovery performance from link and/or device failures that

may happen due to crashes or battery exhaustion of devices or to temporary obstacles. W*-Grid guarantees, by construction, at least two disjoint paths between each couple of nodes. This implies that the recovery in W*-Grid occurs without broadcasting transmissions and guaranteeing robustness while drastically reducing the energy consumption.

An extensive number of simulations shows the efficiency, robustness and traffic load of resulting networks under several scenarios of device density and of number of coordinates. Performance analysis have been compared to existent algorithms in order to validate the results.

Acknowledgements

A special thanks to my advisors Prof. Claudio Sartori and Prof. Gianluca Moro for the time they spent and the great research experience they shared with me in technical discussions, advises and help during these three years I spent as a Ph.D. Student at DEIS.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent advances in information communication technology have led to the rapid development of small, powerful, multi-function devices with multi standard radio interfaces including Bluetooth, Wi-Fi and Wi-Max. For example, ad-hoc networks are being designed where devices/nodes can directly communicate within a limited space both indoor, such as a building, and outdoor, such as a metropolitan area, without the need of a fixed pre-configured infrastructure and rigid data/communication protocols. These wireless ad-hoc networks of computers, sensors, PDAs etc. (i.e. nodes) are revolutionizing connectivity and leading to a paradigm shift from centralized systems to highly distributed and dynamic environments.

Compared to wired networks, wireless networks have unique characteristics. In wireless networks, nodes failure may cause frequent network topology changes, which are rare in wired networks. In contrast to the stable link capacity of wired networks, wireless link capacity continually varies because of the impacts from transmission power, receiver sensitivity and interference. Additionally, wireless networks have power restrictions and bandwidth limitations.

Wireless networks can be classified into infrastructure networks and ad hoc networks according to their dependence on fixed infrastructures. In an infrastructure network, nodes have wired access points (or base stations) within their transmission range. The access points compose the backbone for an infrastructure network. In contrast, wireless ad hoc networks are autonomously self-organized networks without infrastructure support. In a wireless ad hoc network the network may experiences rapid and unpredictable topology changes because nodes in a wireless ad hoc network normally have limited transmission ranges, some nodes cannot communicate directly with each other. Hence, routing paths in ad hoc networks potentially contain multiple hops, and every node has the responsibility to act as a router. Hence, the goal is to enable self-organizing ad-hoc networks, composed of wireless devices including sensors, which are virtually free from configuration and administration costs, and to support location and time sensitive applications in variety of domains. Wireless ad hoc networks are appropriate for applications either in hostile environments where no infrastructure is available, or temporarily established applications which are cost crucial. In recent years, application domains of ad hoc networks gain more and more importance in non-military public organizations and in commercial and industrial areas. The typical application scenarios include the rescue missions, the law enforcement operations, the cooperating industrial robots, the traffic management, and the educational operations in campus.

A plethora of routing algorithms have been proposed for the network path discovery ranging from broadcasting/flooding-based approaches to those using global positioning systems (GPS) to discover the routing path towards the destination. Broadcast algorithms, while simple to implement, are not scalable due to the enormous overhead caused by congestion in large networks. On the other hand, solutions based on GPS, which rely on exact geographic position for each node, does not work in indoor environments and does not function correctly in extremely dense networks or in adverse climatic conditions. Technical and economic feasibility constraints also prevent from attaching a GPS receiver to each node in very large network (i.e. made of thousand of devices). For these reasons our solution does not rely on GPS or any other positioning system. The routing problem has also been addressed in cases of both total absence and partial availability of geographic location information by generating virtual coordinates to approximate real ones.

Our solution may be classified within this set of approaches in that it also uses virtual coordinates, but it is distinctive in that it does not aim to approximate real coordinates. We propose a novel decentralized infrastructure that self-organizes wireless devices in an ad-hoc network, where each node has one or more virtual coordinates through which both message routing and data management occur without reliance on either flooding/broadcasting operations or GPS. The resulting ad-hoc network does not suffer from the dead-end problem, which happens in geographic-based routing when a node is unable to locate a neighbor closer to the destination than itself.

The W-Grid generates, in decentralized manner, virtual coordinates for each network device which reflect its local connectivity with other devices and uses this information to support message routing. These virtual coordinates also delineate the data space partition for which a device is assigned management responsibility, meaning that it is possible to distribute across the W-Grid network any kind of data. In order to proof this feature we will give a short description of a location service. Basically W-Grid [21] [17] is a binary tree index cross-layering both routing and data management features, in that (1) it allows efficient message routing and, at the same time, (2) the virtual coordinates determine a data indexing space partition for the management of multi-dimensional data. Each node has one or more virtual coordinates on which the order relation is defined and through which the routing occurs, and each virtual coordinate represents a portion of the data indexing space for which a device is assigned the management responsibility. Differently from algorithms based on geographic routing (see Chapter 2), W-Grid routing is not affected by dead-ends. To proof the routing and multi-dimensional data management features we will give a short description of a location service in which finding the location of a specific device reduces to a query over a distributed database.

W-Grid can also simply act as the routing network layer upon which existing indexing structures can be applied. For instance we think about the ones that were developed in the past for centralized environments (e.g. [7] and [26], see [5] for an extensive survey) and which have been extended in the last years to work in distributed environments, especially in wired overlay peer-to-peer networks [30] [33] [32] on top of TCP/IP layer of well-organized physical networks.

The multi-dimensional data management capability will be described showing, as an example, how the location service reduces to a simple query, like for any other data type. Extensive performance analysis and experiments have been conducted and the results compared to GPSR, which is considered the most efficient routing solution not using broadcast operations. Our approach shows significant performance gains.

We consider W-Grid to be used in wireless ad-hoc and sensor networks where, though nodes are not inherently mobile, each device can also disconnect from the network (e.g. failures).

1.1 W^R-Grid: Data replication in W-Grid

Since large-scale sensor networks would be expected to serve a substantial number of queries simultaneously for several applications (e.g. humidity, temperature, light etc. for weather monitoring application; temperature, light, presence of chemicals etc. for precision agriculture application and so on.), it has been proven that multi-dimensional data indexing structure can greatly improve query processing efficiency [15]. Data indexing can efficiently work if there is an underlying level of the network performing physical routing without propagating each message to the entire network. The routing service could exploit Global Positioning System (GPS), however, due to its high cost, huge power consumption and unavailability in some environments, GPS is not always a good solution for sensor networks. In fact, in environments where the satellite signal can be obstructed or in indoor environments, the GPS device is unable to provide localization and, consequently, the routing. W^R-Grid extends the infrastructure developed in [19] with data replication. The infrastructure allows multi-dimensional data management and routing, and it is based on the generation and indexing of *virtual coordinates*. The replication strategy offers improvements and new features with respect to the preceding solution. As will be illustrated in the experimental results the replication reduces the average number of hops in the network up to 50%, improving significantly both the energy consumption and the workload balancing among sensors.

1.2 W*-Grid: W-Grid for Sensor networks

Recent improvements in the memory and computational power of sensors, together with the reduction of energy consumptions, are moving researches towards the development of data-centric sensor networks. In this kind of networks nodes are smart enough either to store some data and to perform basic processing allowing the network itself to supply higher level information closer to the network user expectations. In other words, sensors no longer transmit each elementary data sensed, rather they cooperate in order to assemble them in more complex and synthetic information, which will be locally stored and transmitted according to queries and/or events defined by users and external applications.

Energy saving in wireless sensor networks is essential due to limitations in battery lifetime in both MAC and network layers; the routing protocol should avoid complex evaluations of possible paths and should require a minimal knowledge of the network organization. The W-Grid [19] [18] routing scheme satisfies both previously described requirements, in fact its routing protocol needs information about only one-hop away devices and the choice of the next hop requires a bit-a-bit comparison of simple binary strings. In order to improve sensors' failure robustness we implemented W*-Grid. In W*-Grid, whenever a sensor or a link crash or turns off, neighbor sensors are able to recover [19] the network failure without any broadcasting. In this way the network is able to tolerate an arbitrary number of single disconnections.

The solution allows network recovery without ever broadcasting/flooding messages, just because, in principle, it is very expensive and difficult to control. The approach is based on a novel decentralized technique for the assignment of virtual coordinates to nodes that guarantees, by construction, at least two disjoint paths between each couple of nodes, namely two walks without a common node. This innovation drastically reduces the network traffic, while guaranteeing robustness and efficiency. Of course when the network is partitioned or when two subnetworks are just connected thanks to a single link, it is physically impossible to guarantee two disjoint paths between any couple of nodes.

Chapter 2

Related Works

An important and essential issue for wireless ad hoc networks is routing protocol design that is a major technical challenge due to the dynamism of the network. Routing is a fundamental issue for any networks. A lot of routing algorithms have been proposed for wired networks and some of them have been widely used. Dynamic routing approaches are prevalent in wired networks. Distance Vector routing [34] and Link State routing [34] are two of the most popular dynamic routing algorithms used in wired networks. In Distance Vector routing, every router maintains a routing table (i.e. vector), in which it stores the distance information to all reachable destinations. A router exchanges distance information with its neighbors periodically to update its routing table. Routing Information Protocol (RIP) [9] is based on Distance Vector Routing. In Link State routing algorithm, each node periodically notifies its current status of links to all routers in the network. Whenever a link state change occurs, the respective notifications will be flooded throughout the whole network and all routers must re-compute their routes according to the new topology information. In this way, a router gets to know at least a partial picture of the whole network. While in wired networks, Distance Vector and Link State routing algorithms perform well, however, the dynamicity of ad hoc networks affect their functionality. In mobile ad hoc networks, when using a Distance Vector routing or Link State based routing protocol designed for wired networks, frequent topology changes will greatly increase the control overhead. Without remedy, the overhead may overuse scarce bandwidth of mobile ad hoc networks.

2.1 Routing protocols

One of the most popular method to distinguish wireless ad hoc network routing protocols is based on how routing information is acquired and maintained by nodes. Thus, routing protocol for ad-hoc networks are typically subdivided into two main categories: Table-driven (also known as proactive) and On-Demand (or Reactive).

2.1.1 Table-driven Routing Protocol

In a proactive routing protocol nodes participating in the Table-driven network continuously evaluate routes to all reachable nodes so that a source node can get a routing path immediately if it needs one. In these routing protocols nodes need to maintain a consistent view of the network topology and whenever the network topology changes, updates must be propagated to notify the change. Most proactive routing protocols for ad hoc networks inherit properties of wired networks ones but, in order to adapt to the dynamic features of wireless ad hoc networks some modifications have been made. Since in these routing algorithms, wireless nodes proactively update network state and maintain a route regardless of whether data traffic exists or not, the overhead to maintain up-to-date network topology information is high. Examples of proactive routing protocols are the Destination Sequence Distance Vector (DSDV) [27] and the Wireless Routing Protocol (WRP) [24].

The Destination Sequence Distance Vector (DSDV) routing protocol. The Destination Sequence Distance Vector (DSDV) [27] is a proactive unicast wireless ad hoc network routing protocol. Like WRP, DSDV is also based on the traditional Bellman-Ford algorithm. However, their mechanisms to improve routing performance in wireless ad hoc networks are quite different. In routing tables of DSDV, an entry stores the next hop towards a destination, the cost metric for the routing path to the destination and a destination sequence number that is created by the destination. Sequence numbers are used in DSDV to distinguish stale routes from fresh ones and avoid formation of route loops. The route updates of DSDV can be either time-driven or event-driven. Every node periodically transmits updates including its routing information to its immediate neighbors. While a significant change occurs from the last update, a node can transmit its changed routing table in an event-triggered style.

The Wireless Routing Protocol (WRP). The Wireless Routing Protocol [24] is a proactive unicast routing protocol for wireless ad hoc networks. WRP uses improved Bellman-Ford Distance Vector routing algorithm. To adapt to the dynamic features of wireless ad hoc networks, some mechanisms are introduced to ensure the reliable exchange of update messages and reduces route loops. Using WRP, each node maintains a distance table, a routing table, a link-cost table and a Message Retransmission List (MRL). An entry in the routing table contains the distance to a destination node, the predecessor and the successor along the paths to the destination, and a tag to identify its state, i.e., is it a simple path, a loop or invalid. Storing predecessor and successor in the routing table helps to detect routing loops and avoid counting-to-infinity problem, which is the main shortcoming of the original distance vector routing algorithm. A node creates an entry for each neighbor in its link-cost table. The entry contains cost of the link connecting to the neighbor, and the number of timeouts since an error-free message was received from that neighbor. In WRP, nodes exchange routing tables with their neighbors using update messages. The update messages can be sent either periodically or whenever link state changes happen. On receiving an update message, the node modifies its distance table and looks for better routing paths according to the updated information. In WRP, a node checks the consistency of its neighbors after detecting any link change. A consistency check helps to eliminate loops and speed up convergence. One shortcoming of WRP is that it needs large memory storage and computing resource to maintain several tables. Moreover, as a proactive routing protocol, it has a limited scalability and is not suitable for large ad hoc networks.

2.1.2 Reactive routing protocols

Reactive routing protocols for wireless ad hoc networks are also called ”on-demand” routing protocols. In a reactive routing protocol, routing paths are searched only when needed. A route discovery operation invokes a route-determination procedure. The discovery procedure terminates either when a route has been found or no route available after examination for all route permutations. In a wireless ad hoc network, active routes may be disconnected, therefore, route maintenance is an important operation. Compared to the proactive routing protocols for ad hoc networks, less control overhead is a distinct advantage of the reactive routing protocols. Thus, reactive routing protocols have better scalability than proactive routing protocols in wireless ad hoc networks. However, when using reactive routing protocols, source nodes may suffer from long delays for route searching before they can forward data packets. The Dynamic Source Routing (DSR) [12] and Ad hoc On-demand Distance Vector routing (AODV) [28] are examples for reactive routing protocols for ad hoc networks.

The Dynamic Source Routing (DSR) Protocol. The Dynamic Source Routing (DSR) [12] is a reactive unicast routing protocol that utilizes source routing algorithm. In source routing algorithm, each data packet contains complete routing information to reach its dissemination. Additionally, in DSR each node uses caching technology to maintain route information that it has learnt. There are two major phases in DSR, the route discovery phase and the route maintenance phase. When a source node wants to send a packet, it firstly consults its route cache. If the required route is available, the source node includes the routing information inside the data packet before sending it. Otherwise, the source node initiates a route discovery operation by broadcasting route request packets. A route request packet contains addresses of both the source and the destination and a unique number to identify the request. Receiving a route request packet, a node checks its route cache. If the node does not have routing information for the requested destination, it appends its own address to the route record field of the route request packet. Then, the request packet is forwarded to its neighbors. To limit the communication overhead of route request packets, a node processes route request packets that both it has not seen before and its address is not presented in the route record field. If the route request packet reaches the destination or an intermediate node has routing information to the destination, a route reply packet is generated. When the route reply packet is generated by the destination, it comprises addresses of nodes that have been traversed by the route request packet. Otherwise, the route reply packet comprises the addresses of nodes the route request packet has traversed concatenated with the route in the intermediate nodes route cache. After being created, either by the destination or an intermediate node, a route reply packet needs a route back to the source. There are three possibilities to get a backward route. The first one is that the node already has a route to the source. The second possibility is that the network has symmetric (bi-directional) links. The route reply packet is sent using the collected routing information in the route record field, but in a reverse order as shown in Figure 1. In the last case, there exists asymmetric (uni-directional) links and a new route discovery procedure is initiated to the source. The discovered route is piggybacked in the route request packet. In DSR, when the data link layer detects a link disconnection, a ROUTE_ERROR packet is sent backward to the source. After receiving the ROUTE_ERROR packet, the source node initiates another route discovery operation. Additionally, all routes containing the broken link should be removed from the route caches of the immediate nodes when the ROUTE_ERROR packet is transmitted to the source. DSR has increased traffic overhead by containing complete routing information into each data packet, which degrades its routing performance.

The Ad Hoc On-demand Distance Vector Routing (AODV) protocol. The Ad Hoc On-demand Distance Vector Routing (AODV) protocol [28] is a reactive unicast routing protocol for ad hoc networks. As a reactive routing protocol, AODV only needs to maintain the routing information about the active paths. In AODV, routing information is maintained in routing tables at nodes. Every node keeps a next-hop routing table, which contains the destinations to which it currently has a route. A routing table entry expires if it has not been used or reactivated for a pre-specified expiration time. Moreover, AODV adopts the destination sequence number technique used by DSDV in an on-demand way. In AODV, when a source node wants to send packets to the destination but no route is available, it initiates a route discovery operation. In the route discovery operation, the source broadcasts route request (RREQ) packets. A RREQ includes addresses of the source and the destination, the broadcast ID, which is used as its identifier, the last seen sequence number of the destination as well as the source nodes sequence number. Sequence numbers are important to ensure loop-free and up-to-date routes. To reduce the flooding overhead, a node discards RREQs that it has seen before and the expanding ring search algorithm is used in route discovery operation. The RREQ starts with a small TTL (Time-To-Live) value. If the destination is not found, the TTL is increased in following RREQs.

2.1.3 Hybrid routing protocols

Hybrid routing protocols are proposed to combine the merits of both proactive and reactive routing protocols and overcome their shortcomings. Normally, hybrid routing protocols for wireless ad hoc networks exploit hierarchical network architectures. Proper proactive routing approach and reactive routing approach are exploited in different hierarchical levels, respectively. An example of hybrid routing protocols for wireless ad hoc networks is the Zone Routing Protocol (ZRP).

The Zone Routing Protocol (ZRP). The Zone Routing Protocol (ZRP) [8] is a hybrid routing protocol for ad hoc networks. The hybrid protocols are proposed to reduce the control overhead of proactive routing approaches and decrease the latency caused by route search operations in reactive routing approaches. In ZRP, the network is divided into routing zones according to distances between nodes. Given a hop distance d and a node N, all nodes within hop distance at most d from N belong to the routing zone of N. Peripheral nodes of N are Ns neighboring nodes in its routing zone which are exactly d hops away from N. In ZRP, different routing approaches are exploited for inter-zone and intra-zone packets. The proactive routing approach, i.e., the Intra-zone Routing protocol (IARP), is used inside routing zones and the reactive Inter-zone Routing Protocol (IERP) is used between routing zones, respectively. The IARP maintains link state information for nodes within specified distance d. Therefore, if the source

2.1.4 Geographic routing protocols

A completely different approach is used by geographic routing protocols such as [13] [14]. The idea in geographical routing is to use a node's location as its address, and to forward packets with the goal of reducing as much as possible the physical distance to the destination. Geographic routing achieves good scalability since each node only needs to be aware of neighbors' position and because it does not rely on flooding to exploit network topology. However it suffers of dead end problems, especially under low density environment or scenarios with obstacles or holes. Problems are caused by the inherent greedy nature of the algorithm that can lead to situation in which a packet gets stuck at a local optimal node that appears closer to the destination than any of its known neighbors. In order to solve this flaw, correction methods such as perimeter routing, that tries to exploit the well-known right hand rule, have been implemented. However, some packet losses still remain and furthermore using perimeter routing causes loss of efficiency both in terms of average path length and of energy consumption. Another limitation of geographic routing is that it needs nodes to know their physical position. Usually authors assume that they embed GPS but it must be said that GPS receivers are expensive and energy inefficient compared to the devices that could participate in ad-hoc networks. Besides, GPS reception might be easily obstructed by climatic conditions or obstacles and doesn't work indoor.

Recently, virtual coordinates were proposed to exploit the advantages of geographic routing in absence of location information [29] [23] [2]. The motivation is that in many applications it is not necessary to know the exact coordinates but is often sufficient to have virtual coordinates that approximate real ones. Unfortunately virtual coordinate systems suffer the same dead end problem of standard geographic routing. W-Grid employs virtual coordinates like these last algorithms but it is based on a different approach which does not approximate real coordinates and eliminates the risk of dead-ends.

2.2 Network Structure Organization

Another classification method is based on the roles which nodes may have in a routing scheme. In a uniform routing protocol, all nodes have same role, importance and functionality. Examples of uniform routing protocols include Wireless Routing Protocol (WRP), Dynamic Source Routing (DSR), Ad hoc On-demand Distance Vector routing (AODV) and Destination Sequence Distance Vector (DSDV) routing protocol. Uniform routing protocols normally assume a flat network structure. In a non-uniform routing protocol for ad hoc networks, some nodes carry out distinct management and/or routing functions. Normally, distributed algorithms are exploited to select those special nodes. In some cases, non-uniform routing approaches are related to hierarchical network structures to facilitate node organization and management. Non-uniform routing protocols further can be divided according to the organization of nodes and how management and routing functions are performed. Following these criteria, non-uniform routing protocols for ad hoc networks are divided into zone based hierarchical routing; cluster-based hierarchical routing and core-node based routing. In zone based routing protocols, different zone constructing algorithms are exploited for node organization, e.g some zone constructing algorithms uses geographical information. Also zones may overlap or not depending on the constructing method. Exploiting zone division effectively reduces the overhead for routing information maintenance. Mobile nodes in the same zone know how to reach each other with smaller cost compared to maintaining routing information for all nodes in the whole network. In some zone based routing protocols, specific nodes act as gateway nodes and carry out inter-zone communication. The Zone Routing Protocol (ZRP) is a zone based hierarchical routing protocols for ad hoc networks. A cluster based routing protocol uses specific clustering algorithm for clusterhead election. Mobile nodes are grouped into clusters and clusterheads take the responsibility for membership management and routing functions. Clusterhead Gateway Switch Routing (CGSR) [3] will be introduced in Section 5 as an example of cluster based wireless ad hoc network routing protocols. Some cluster based ad hoc network routing protocols potentially support a multi-level cluster structure, such as the Hierarchical State Routing (HSR) [11]. In core-node based routing protocols for ad hoc networks, critical nodes are dynamically selected to compose a "backbone" for the network. The backbone nodes carry out special functions, such as routing paths construction and control/data packets propagation.

The Clusterhead Gateway Switch Routing (CGSR). The Clusterhead Gateway Switch Routing (CGSR) [3] is a hierarchical routing protocol. The cluster structure improves performance of the routing protocol because it provides effective membership and traffic management. Besides routing information collection, update and distribution, cluster construction and clusterhead selection algorithms are important components of cluster based routing protocols. CGSR uses similar proactive routing mechanism as DSDV. Using CGSR, nodes are aggregated into clusters and a cluster-head is elected for each cluster. Gateway nodes are responsible for communication between two or more clusterheads. Nodes maintain a cluster member table that maps each node to its respective cluster-head. A node broadcasts its cluster member table periodically. After receiving broadcasts from other nodes, a node uses the DSDV algorithm to update its cluster member table. In addition, each node maintains a routing table that determines the next hop to reach other clusters. In a dynamic network, cluster based schemes suffer from performance degradation due to the frequent elections of a clusterhead. To improve the performance of CGSR, a Least Cluster Change (LCC) algorithm is proposed. Only when changes of network topology cause two clusterheads merging into one or a node being out of the coverage of all current clusters, LCC is initiated to change current state of clusters. In CGSR, when forwarding a packet, a node firstly checks both its cluster member table and routing table and tries to find the nearest clusterhead along the routing path.

2.3 Data Management

With regard to MAC protocol for wireless sensor networks we can distinguish existing solutions in two main categories. The first category includes IEEE 802.11 protocol [1] and protocols based on it. The main problem with IEEE 802.11 is that it consumes energy by continuous idle listening. For this reason proposals such as S-MAC [37] and T-MAC [38] try to reduce different sources of energy consumption, for instance by limiting overhearing or by using periodic sleeping and listening (802.11 Power Saving mode) to reduce idle listening. Another category is represented by TDMA-based protocols, however, since these protocols require centralized control of nodes, they are not suitable for the type of networks we want to manage. However, also the previously described variations of IEEE 802.11 require a certain coordination among nodes in order to define sleep periods and usually this coordination is given by a sink node with particular tasks different from the rest of the network. As a result also this kind of protocols are not applicable. Existing routing protocols have been developed by following different approaches.

Basically routing is necessary whenever a data sensed (we also say generated) must be transmitted elsewhere in the network, including an external machine, proactively or reactively according to periodic tasks or queries submitted to the network system. As stated before, we do not consider sensor networks which simply transmit data externally at a remote base station, we focus on advances wireless sensor networks in which data or events are kept at sensors, are indexed by attributes and represented as relations in a virtual distributed database. For instance in [15, 10, 36], data generated at a node is assumed to be stored at the same node, and queries are either flooded throughout the network [10].

In a GHT [31], data is hashed by name to a location within the network, enabling highly efficient rendezvous. GHTs are built upon the GPSR [13] protocol and leverage some interesting properties of that protocol, such as the ability to route to a sensors nearest to a given location, together with some of its limits, such as the risk of dead ends. Dead end problems, especially under low density environment or scenarios with obstacles or holes, are caused by the inherent greedy nature of the algorithm that can lead to situation in which a packet gets stuck at a local optimal sensors that appears closer to the destination than any of its known neighbors. In order to solve this flaw, correction methods such as perimeter routing, that tries to exploit the right hand rule, have been implemented. However, some packet losses still remain and furthermore using perimeter routing causes loss of efficiency both in terms of average path length and of energy consumption. Besides, another limitation of geographic routing is that it needs sensors to know their physical position adding localization costs to the system. In DIFS [6], Greenstein et al. have designed a spatially distributed index to facilitate range searches over attributes.

Like us, in [15] and [35] authors have built a distributed index for multi-dimensional range queries of attributes but they require nodes to be aware of their physical location and of network perimeter; moreover they exploit GPSR for routing which is subjected to dead-ends and loss of packets. Our solution also behaves like a distributed index, but its indexing feature is cross-layered with routing, meaning that no physical position nor any external routing protocol is necessary, routing information is given by the index itself. In [15] and [35] data space partitions follow the physical positions of nodes, which means that even if data are uniformly distributed in the multi-dimensional space (ideal condition) the storage load per node is, in general, unbalanced, because it depends on the physical network topology; this leads to an unbalanced energy consumption among nodes and consequently to a rapid network break-up caused by premature turning off of most loaded sensors. In W-Grid the storage load balancing has been achieved thanks to two key points: (i) the multi-dimensional data space partitions occur according to the actual data distribution and (ii) each partions has the same maximum bucket size. Besides, data partitions in [15] and [35] are disjoint, while in W-Grid they are nested.

As in peernet [4] our virtual coordinates are binary strings, however, our coordinate generation method does not need to define a priori a coordinate length. This means that in W-Grid it is always possible to assign new coordinates when new nodes join the network. Besides, we do not impose only one coordinate per node because this increases both the risk of unbalanced networks and the average number of hops. Finally peernet is not designed to manage, index and querying distributed multi-dimensional data.

Chapter 3

W-Grid

The main idea is to map nodes on a binary tree so that the resulting coordinate space reflects the underlying connectivity among them. Basically we aim to set parent-child relationships to the nodes which can sense each other, in this way we are always able to route messages, in the worst cases simply following the paths indicated by the tree structure. Using virtual coordinates that do not try to approximate node's geographic position we eliminate any risk of dead-ends.

We consider the case of nodes equipped with a wireless device. Each one is, at the same time, client of the network (e.g. sending messages, request services), responsible for managing others nodes communications (e.g. routing and forwarding messages) and supplier of information and services. For this reason from now on we will refer to them as nodes, sensors or peers indistinctly.

Basically W-Grid can be viewed as a binary tree index cross-layering both routing and data management features in that, (1) by implicitly generating coordinates and relations among nodes allows efficient message routing and, at the same time, (2) the coordinates determine a data indexing space partition for the management of multi-dimensional data. Each node has one or more virtual coordinates on which the order relation is defined and through which the routing occurs, and at the same time each virtual coordinate represents a portion of the data indexing space for which a device is assigned the management responsibility. W-Grid virtual coordinates are generated on a one-dimensional space and the devices do not need to have knowledge of their physical location. Thus, differently from algorithms based on geographic routing (see chapter 2), W-Grid routing is not affected by dead-ends. Since in sensor networks the most important operations are data gathering and querying it is necessary to guarantee the best efficiency during these tasks.

In the next sections we will introduce a formal description of the main W-Grid features.

3.1 Virtual Coordinates Generation

When a node, let us say n turns on for the first time, it starts a wireless channel scan (beaconing) searching for any existing W-Grid network to join (namely any neighbor device that already holds W-Grid virtual coordinates). If none W-Grid network is discovered, n creates a brand new virtual space coordinate and elects itself as root by getting the virtual coordinate $" * "$[1]. On the contrary, if beaconing returns one or more devices which hold already a W-Grid coordinate, n will join the existing network by getting a virtual coordinate.

Coordinate Setup. Whenever a node needs a new W-Grid coordinate, an existing one must be split. The term "split" may seem misleading at the moment, but its meaning will become straightforward clear in Section 3.6. A new coordinate is given by an already participating node n_g, and we say that its coordinate c is split by concatenating a 0 or a 1 to it. The result of a split to c will be $c' = c + 1$ and $c'' = c + 0$. Then, one of the new coordinates is assigned to the joining node, while the other one is kept by the giving node. No more splits can be performed on the original coordinate c since this would generate duplicates. In order to guarantee coordinates' univocity even in case of simultaneous requests, each asking node must be acknowledged by the giving one n_g. Thus, if two nodes ask for the same coordinate to split, only one request will succeed, while the other one will be canceled.

Coordinate Selection. At coordinate setup, if there are more neighbors which already participate the W-Grid network, the joining sensor must choose one of them from which to take a coordinate. The selection strategy we adopt is to choose the shortest coordinate[2] in terms of number of bits. If two or more strings have the same length the sensor randomly chooses one of them. Experiments have shown that this policy of coordinate selection reduces as much as possible the average coordinates length in the system.

In Figure 3.1 there is a small example of a W-Grid network. In the tree structure, parent-child relationships can be set only by nodes that are capable of bi-directional direct communication. This property is called *integrity* of coordinates and it is crucial for the network efficiency:

---

[1] It is conventional to label $" * "$ the root node

[2] among the ones that still can be split, see Coordinate Setup

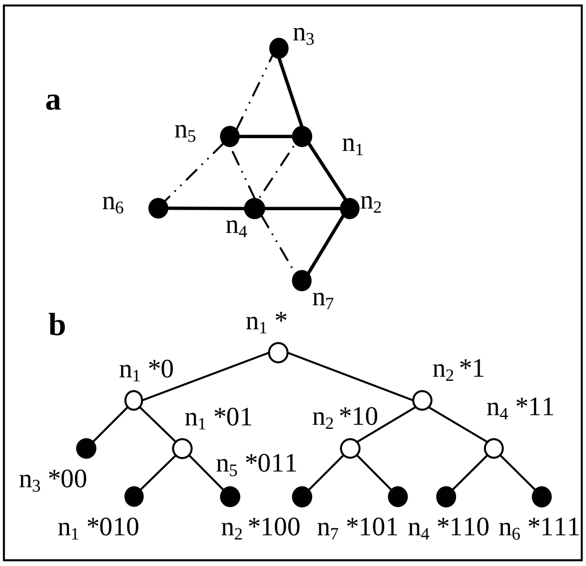


Figure 3.1: Physical (a) and logical (b) network. Empty circles represent split coordinates, full black circles are coordinates that can still be split.

Definition 1 *Let c be a coordinate at node n that has been split into* c' *and* c''*. Then we say that c is integral if* c' *or* c'' *is held by a node* $n' \in NEIGH(n)$*, where* $NEIGH(n)$ *is the set of its neighbors.*

If each coordinate satisfies this constraint, it will be possible to route any message, at least by following the paths indicated by the tree structure, without dead-ends.

3.2 Assigning Multiple Coordinates to Peers

Nodes progressively get new coordinates from their physical neighbors in order to establish parentships with them. The number of coordinates at nodes may vary, in W-Grid that measure is always used as a parameter. The policies for coordinates may be: (1) a fixed number of coordinates per node (e.g. a given k) or (2) one coordinate per physical neighbor. Extensive experiments have showed that assigning different coordinates per node improves routing efficiency, in fact having more than one coordinate means that a node is placed in different positions of the tree structure and this has two positive effects on the system.

Firstly, the probability that two nodes physically close have very different virtual coordinates, which may happen when a multi-dimensional space (in

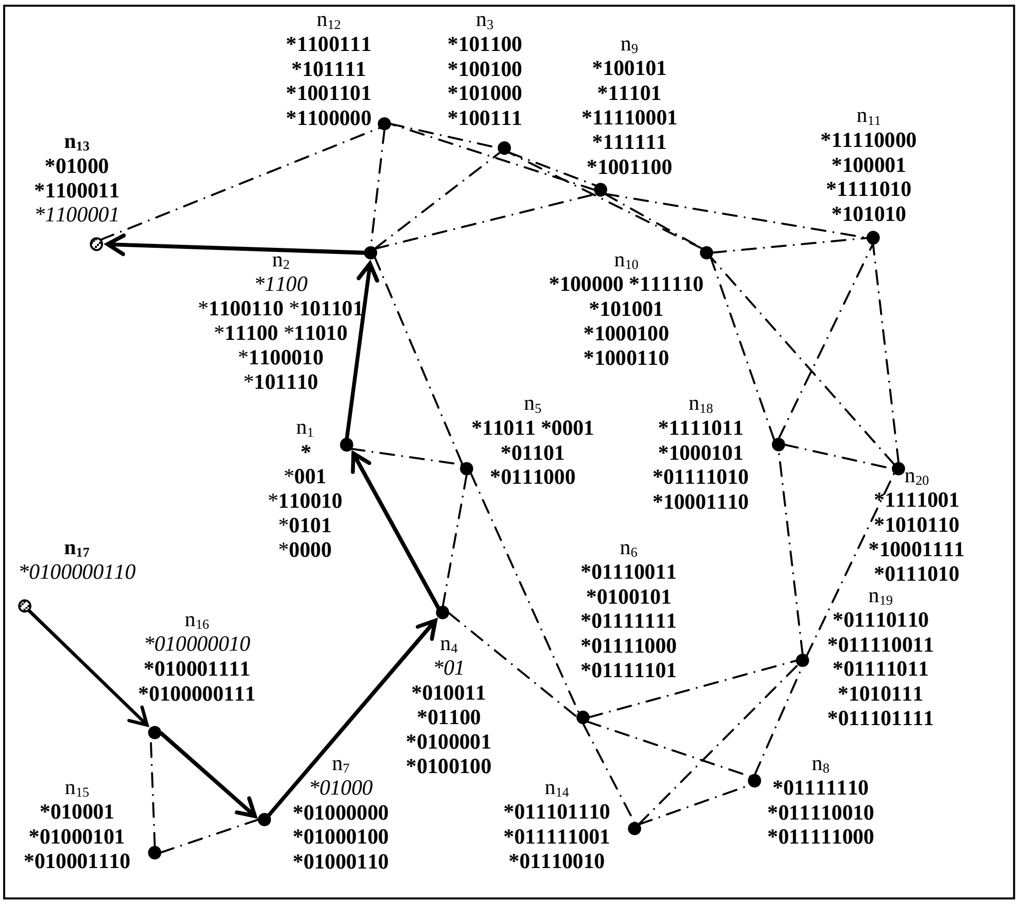


Figure 3.2: A small example of a network with W-Grid coordinates and routing of a message (from node n_{17} to node n_{13}).

which nodes are spread) is mapped into a mono-dimensional space, is highly reduced. Besides, this implies that for each couple of nodes there will be several different paths that allow packet routing, improving network robustness against unexpected failures of nodes. During the coordinate setup, if the number of neighbors holding virtual coordinates is more than one, let us say k, n_j must choose one node among $n_1, .., n_k$ and ask for a coordinate. The selection strategy we adopt is to choose the shortest coordinate (in terms of number of bits). If two or more strings have the same lengths the nodes will choose the one that is more distant from all the other candidates. The choice of the shortest possible c aims to reduce as much as possible the length of the coordinates in the system.

In W-Grid we map a multi-dimensional space in a one dimension space.

Whenever the number of dimensions of a space is reduced, some points of the space lose proximity. Since W-Grid virtual coordinates space is one-dimensional, while nodes are spread on a two-dimensional space (for simplicity consider nodes to at the same height), it means that two nodes physically close in the real space can be far away in the virtual space (e.g. they have very different virtual coordinates). As routing is performed through virtual coordinates surely it will lose efficiency whenever these situations occurs. We came to the conclusion that it is possible to widely reduce inefficiencies be assigning more different coordinates to each node. In fact, having more than one coordinate means that a node is placed in different positions of the tree structure and reduces the probability that two nodes physically close are very distant according to the order relation.

In Figure 3.2 each node is assigned a number of virtual coordinates equal to the number of their neighbors. Simulations returned that this coordinates generation policy ensures the best results in terms of combination between network efficiency and quantity of information stored at nodes. In fact there is a trade-off between these two measures since a higher number of coordinates per node translates into best routing performances but also implies larger routing tables and needs more storage capability at nodes.

In order to improve readability of the figure, for each node are shown only the coordinates that have not been split. The only exceptions are the coordinates interested by routing from node n_{17} to node n_{13}. This in useful to understand that split coordinates are stored at nodes and are used for routing. For instance node n_1, the root of the coordinate space, holds also coordinates $*$, $*0$ and $*00$; namely through multiple splits of root coordinate $*$ we obtained $*001$.

3.3 Formal Model

The sensor network is represented as a graph S:

$$S = (D, L)$$

in which D is the set of participating devices and L is the set of physical connectivity between couples of devices:

$$L = \{(d_i, d_j) : two - way\ connection\ between\ d_i\ and\ d_j\}$$

Each device is assigned one or more (virtual) coordinate(s). We define C as the set of existing coordinates. Each coordinate c_i is represented as a string of bits starting with $\star$. According to the regular expression formalism coordinates are defined as follows:

$$C = \{c : \; c = \star(0 \mid 1)^*\}$$

E.g. $\star 01001$ is a valid W-Grid coordinate. Given a coordinate c_i and a bit b their concatenation will be indicated as $c_i b$. E.g. considering $c_i = \star 0100, b = 0$ then $c_i b = \star 01000$. Given a bit b its complementary $\overline{b}$ is defined. E.g $\overline{1} = 0$.

Some functions are defined on C:

$$length(c) : C \rightarrow \; \mathbb{N} \tag{3.1}$$

Given a coordinate c, $length(c)$ returns the number of bits in c. ($\star$ excluded). E.g. $length(\star 01001) = 5$.

$$bit(c, k) : (C, \mathbb{N} - \{0\}) \rightarrow \; \{0, 1\} \tag{3.2}$$

Given a coordinate c and a positive integer $k \leq length(c)$, $bit(c, k)$ returns the k-th bit of c. Position 0 is out of the domain since it is occupied by $\star$.

$$pref(c, k) : (\mathbb{C}, \mathbb{N}) \rightarrow \; C \tag{3.3}$$

Given a coordinate c and a positive integer $k \leq length(c)$, $pref(c, k)$ returns the first k bits of c. E.g. $pref(\star 01001, 3) = \star 010$. We define the complementary(buddy) of a coordinate c as:

$$\overline{c} = pref(c, length(c) - 1)\overline{bit(c, length(c))} \tag{3.4}$$

E.g. $\overline{\star 01001)} = \star 01000$.

$$father(c) : (C - \{\star\}) \rightarrow \; C$$

$$father(c) = pref(c, length(c) - 1) \tag{3.5}$$

$$lChild(c), rChild(c) : (C) \rightarrow \; C$$

$$lChild(c) = c0 \tag{3.6}$$

$$rChild(c) = c1 \tag{3.7}$$

E.g. Given a coordinate $c_i = \star 011$, $father(\star 011) = \star 01$, $rChild(\star 011) = \star 0111$, $lChild(\star 011) = \star 0110$.

A function M *maps* each coordinate c to the device holding it:

$$M : C \rightarrow D$$

A W-Grid network is represented as a graph:

$$W = (C, P)$$

P is the set of *parentships* between coordinates.

$$P = \{(c_i, c_j) : c_j = c_i(0 \mid 1)\}$$

E.g. $p_i = (\star010, \star0101)$. We define the complementary(buddy) of a parentship $p = (c_i, c_j)$ as:

$$\overline{p} = (c_i, \overline{c_j}) \tag{3.8}$$

E.g. $p = (\star010, \star0101)$, $\overline{p} = (\star010, \star0100)$. A graph W is a valid W-Grid network if all the following properties are satisfied:

1. $\forall p = (c_i, c_j) \in P, (M(c_i) = M(c_j)) \vee ((M(c_i), M(c_j)) \in L)$
2. $\forall p = (c_i, c_j) \in P : M(c_i) \neq M(c_j) \Rightarrow \exists\ \overline{p} = (c_i, \overline{c_j}) \in P : M(c_i) = M(\overline{c}_j)$

3.4 W-Grid dynamic rules

W-Grid network is generated according to this few simple rules:
1. The first node that joins the networks (that initiate a coordinate space) gets the coordinate $\star$. A node that holds a W-Grid coordinate is marked as **active**. A function *last* is defined:

$$last(d) : (D) \rightarrow\ C$$

which returns the last coordinate received by d. If d is **not active** the function returns $\{\emptyset\}$. After the first node, let us say n_1, has joined the network, $last(n_1) = \star$.
2. $\forall\ l = (d_i, d_j) \in L : last(d_i) \neq \{\emptyset\}$ two parentships are generated:

- $p = (last(d_i), c')$: $M(c') = d_j$
- $\overline{p}$

Where $c' = lChild(last(d_i)) \mid rChild(last(d_i))$. Namely c' corresponds to the non-deterministic choice of one of the children of c.

Nodes progressively get new coordinates from their physical neighbors in order to establish parentships with them. The number of coordinates at nodes may vary, in W-Grid that measure is always used as a parameter. The policies for coordinates may be: (1) a fixed number of coordinates per node (e.g. a given k) or (2) one coordinate per physical neighbor. Coordinates getting is also called "split". The actors of the split procedure are an asking node and a giving node. A coordinate c_i is split by concatenating a bit to it and then, one of the new coordinates is assigned to the joining node, while the other one is kept by the giving node. Obviously, an already split coordinate c_i can not be split anymore since this would generate duplicates. Besides, in order to guarantee coordinates' univocity even in case of simultaneous requests, each asking node must be acknowledged by the giving node. Thus, if two nodes ask for the same coordinate to split, only one request will succeed, while the other one will be temporarily rejected and postponed. Coordinate discovering is gradually performed by implicit overhearing of neighbor sensors transmissions.

3.5 Routing algorithm

W-Grid maps nodes on an indexing binary tree T in order to build a totally ordered set over them. Each node of the tree is assigned a W-Grid virtual coordinate (c) which is represented by a binary string and has a value $v(c)$:

$$\forall\, c \in T, v(c) \in C$$

where C is a totally ordered set since:

$$\forall\, c_1, c_2 \in T : c_2 \in l(c_1) \rightarrow v(c_2) < v(c_1)$$

$$\forall\, c_1, c_2 \in T : c_2 \in r(c_1) \rightarrow v(c_2) > v(c_1)$$

where $r(c)$ and $l(c)$ represents the right sub-tree and the left sub-tree of a coordinate $c \in T$ respectively. And:

$$\forall\, c_1, c_2 \in T : F(c_1, c_2) = 0 \rightarrow v(c_1) < v(c_2)$$

$$\forall\, c_1, c_2 \in T : F(c_1, c_2) = 1 \rightarrow v(c_1) > v(c_2)$$

where $F(c_1, c_2)$ is a function that returns the bit of coordinate c_1 at position $i + 1$ where i corresponds to the length of the common prefix between c_1 and c_2. For instance given two coordinates $c_1 = \mathbf{11}0100$ and $c_2 = \mathbf{11}10$, $F(c_1, c_2) = 0$[3] therefore $c_2 > c_1$.

As we stated before, the coordinate creation algorithm of W-Grid generates an order among the nodes and its structure is represented by a binary tree. The main benefit of such organization is that messages can always be

---

[3] While $F(c_2, c_1) = 1$, therefore $F(c_1, c_2) = \overline{F(c_2, c_1)}$

delivered to any destination coordinate, in the worst case by traveling across the network by following parent-child relationship. The routing of a message is based on the concept of distance among coordinates. The distance between two coordinates c_1 and c_2 is measured in logical hops and correspond to the sum of the number of bits of c_1 and c_2 which are not part of their common prefix. For instance:

$$d(\mathbf{*0}011, \mathbf{*0}11) = 5$$

Obviously it may happen that physical hops distance is less then the logical.

Given a message and a target binary string c_t each node n_i forwards it to the neighbor that present the shortest distance to c_t. It is important to notice that each node needs neither global nor partial knowledge about network topology to route messages, its routing table is limited to information about its direct neighbors' coordinates. This means **scalability** with respect to network size.

W-Grid metric has a very interesting feature. Given a virtual coordinate c and a distance d, there are several $c_i \in C$ which are distant d from c. For instance, given $*0011$ and distance 3:

$$\begin{gathered} d(*0011, *0) = 3 \\ d(*0011, *000) = 3 \\ d(*0011, *00100) = 3 \\ \text{etc.} \end{gathered}$$

In general given a coordinate c of length l, the number of coordinates whose distance from c is d is given by:

$$\sum_{\alpha=\max(1,l-d)}^{\max(1,l-1)} 2^{\Delta-1} \quad \textit{where } \Delta = d - (l - \alpha) \tag{3.9}$$

From (3.9) we can say that for each coordinate and distance there exist a set of coordinates at that distance that we call $c(d)$ (*distance set*). Thus, at each hop during the routing, a node s distant d from the destination has at least one neighbor that improves by one the distance (in logical hops) from the destination[4]. However, it is also possible that other neighbors of s belong to $c(d-1)$. This means a certain robustness to nodes failures and also the possibility of adopting specific and changeable policies for routing (for instance by forwarding to the node with most battery power left, in case of more nodes with the same distance from the target).

---

[4] Effects of the integrity of the coordinates

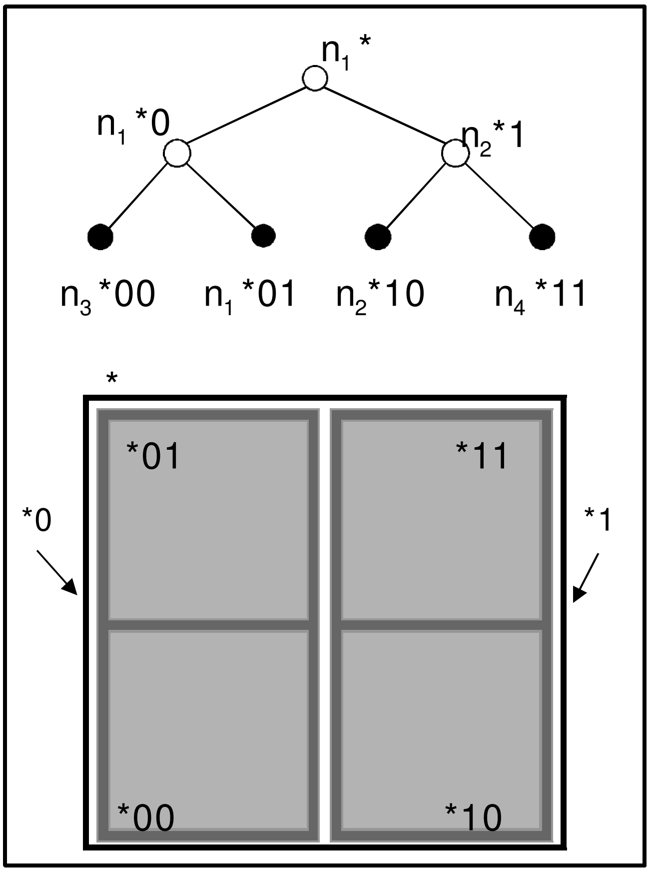


Figure 3.3: Correspondence between coordinates and data space partitions

3.6 Data Management in W-Grid

W-Grid organizes peers in a tree structure and distributes data (tuple or records with any kind of information) among them by hashing the values of the record attributes into binary strings and storing them at peers whose W-Grid coordinates match the strings. Since W-Grid c_i are binary strings, we can see from Figure 3.3 that they correspond to leaf nodes of a binary tree. Therefore a W-Grid network acts directly as a distributed database. This means that each coordinate represent a portion (i.e. region) of the global data space as depicted in Figure 3.3. Regions are generated according to data distribution and the use of a bucket size for each data region, together with a load balancing algorithm, allow to balance nodes storage load [19].

Obviously coordinates that have been split (the empty circles in Figures 3.1 and 3.3) cannot contain any data.

Let us describe a brief example of an environment monitoring application in which sensors survey temperature (T) and pressure (P), to which we refer as d_1 and d_2. Each event is inserted in the distributed database implicitly generated by W-Grid, reporting for instance date and time of occurrence. Without loss of generality we can define a domain for T and P let us say $Dom(d_1) = [-40, 60]$ and $Dom(d_2) = [700, 1100]$. We present two examples: (i) an exact-match and (ii) a range query submitted to the network.

(i) Return the times at which sensors surveyed a temperature of 26 Celsius degrees and a pressure of 1013mbar. The linearization [22, 25] the two-dimensional data values results in a binary string which indicates the path to be followed in the network to get to the sensor storing the data. Then, any sensor can be taken as starting point for the query to get to the destination. In this case the result of the linearization is[5]:

$$c_t = *11011000$$

As described in [22, 25] the length of the destination string can be adjusted, without affecting the hops that were previously covered, during the routing if we find that sensors with longest string exist.

(ii) Return the times at which sensors surveyed a temperature ranging from 26 to 30 Celsius degrees and pressure ranging from 1013 to 1025mbar. After calculating the correspondent binary string for the four corners of the range query, namely:

(26,1013) (26,1025) (30,1013) (30,1025)
$c_1 = *11011000$ $c_2 = *11011001$
$c_3 = *11011010$ $c_4 = *11011011$

all we have to do is querying sensors whose coordinates have $*110110$ as prefix.

One of the most important features that a distributed database must satisfy is a balanced storage load among the different nodes, especially in case of not uniform distributions of data. In fact, if the managed information do not distribute uniformly in the domain space it can happen that virtual coordinates store different number of data. Therefore nodes that manage more data will likely receive a higher number of queries than the others causing bottlenecks and loss of efficiency for the entire network. Due to the coordinates *integrity* constraint, related coordinates must belong to nodes that can directly contact each other. This means that each node can split

---

[5] By standardizing 26 and 1013 to their domains we get 0,76 and 0,78 respectively. We multiply both of them by 2^4 to get a string of length 8. The binary conversion of the multiplications are *1010* and *1100* respectively. Then, by crossing bit by bit the two string we get $*11011000$.

coordinates only a limited number of times, also according to which kind of coordinate creation policy is adopted. However, it is easy to understand that nodes managing shorter coordinates (likely the first nodes joining the network) will split about the same times of any other nodes but with the difference that their initial region are much bigger than the ones of other nodes. It is easy to infer that this translates into a very unbalanced storage load situation.

In order to improve the data distribution balance we implemented the Storage Load Balancing (SLOB) Algorithm that will be described in section 3.7. Then in section 3.7.1 we will show its effects on a real problem, namely the definition of a location service that provides information about the position, yet in terms of W-Grid virtual coordinates, of any participant. Basically, the location service is a usual exact match query on distributed data where there is a correspondence between data and nodes location.

3.7 Storage Load Balancing in W-Grid

To address the load balancing problem, existing in most of data structures that manage multi-dimensional data, we incorporate the concept of bucket size b namely the maximum number of data that a region (i.e. a coordinate) can manage. The value for b can be the same for each peer or, in environments where devices have different characteristics, it can be proportional for instance to the storage and/or communication bandwidth capabilities.

Whenever a node receives a new data it checks wether the space represented by the coordinate that must store the data is full or not. In case it is full the coordinate is split, but, differently from what it happens when a new node joins the network, in this case both the resulting subspaces are stored at the peer.

The bucket size guarantees that each coordinate contains at most the same quantity of information. However, this trick does not balance the storage load on its own. In fact, peers holding spaces with a higher number of data will split more frequently that the others. The result will be that those peers will manage more coordinates if we do not find a way for them to give away the ones in excess, which is exactly the goal of Storage Load Balancing Algorithm (SLOB). On periodic beaconing each peer evaluates the average storage load and the correspondent Root Mean Square Error ($avgNeighLoad$ and $neighLoadRMSE$ in algorithm 1) of its neighbors. The storage load of a node is meant as the number of coordinates held excluding split coordinates (not considered since there can be no data in them).

The purpose of this evaluation is discovering local unbalanced situations

Algorithm 1 Storage LOad Balancing Algorithm

```
$MyLoad \Leftarrow$ storage load at peer
scan neighbors and return $avgNeighLoad$, $neighLoadRMSE$ and $mostLoadedNeighbor$
if $(avgNeighLoad - MyLoad) > avgThreshold\ OR\ (avgNeighLoad > Load\ \&\ RMSE > RMSEThreshold)$ then
  get one $c$ from $mostLoadedNeighbor$
end if
```

and moving a small step towards better balancing. In practice, a peer p_i compares its own load with the average, if the load is lower and the difference between the two measures is higher than a certain threshold (avgThreshold in algorithm 1) p_i takes one coordinate from the neighbor that has the highest storage load. A coordinate is taken anyway if the load is the same as the average but the RMSE is higher than a given threshold (RMSEThreshold in algorithm 1). The algorithm is as much simple as it is powerful since adding a local rule is able to create a global behavior that makes converge the network storage load toward a balanced situation.

3.7.1 Location service

Supposing that each peer n_i that composes the network is univocally identified by a public ID_i (such as the e-mail address, the MAC Address or any other unique ID) we can think about inserting in the distributed database, implicitly defined by W-Grid, information about peers location (W-Grid coordinates) using as key (both for insertion and search) the peers IDs. In this way, a node (n_s) that need to communicate with another node (n_r) simply searches the network for the ID_r and will discover where n_r can be found. After this, n_s will be able to send a message to the recipient simply using the W-Grid routing algorithm.

In order to show W-Grid capability of managing multi-dimensional data we will define the node ID as a pair (prefix,number) where $Dom_{prefix} = [0, 9999]$ and $Dom_{number} = [0, 9999999]$. We use a hashing function (please refer to [20] and [25] for details) to translate IDs into a binary string of arbitrary length.

For instance, if n_s needs to contact the peer n_r identified by $ID_d = (7601, 452789623)$ it can find[6]:

---

[6] By standardizing 7601 and 452789623 to their domains we get 0,76 and 0,45 respectively. We multiply both of them by 2^4 to get a string of length 8. The binary conversion

$$c_d = *10011100$$

1. **The ID_i is scaled into the interval $[0, 1[$.**

$$S(ID_i) = 452789623/1000000000 = 0,452789623$$

2. **The scaled value is multiplied for 2^l.** l corresponds to the desired virtual coordinate length, let us suppose a value of 6 for it

$$0,452789623 * 2^6 = 28,9785$$

3. **The integer part of the calculated value is converted into binary.**

$$28 = 11100$$

4. **The resulting string may need to be extended.** If the length of the string if less than the desired one zeroes are appended on the top of it plus the char "*" which starts every coordinate

***0**011100

$*10011100$ corresponds to the virtual coordinate holding n_d location information, however it is not guaranteed that coordinate actually exists in the network. In fact, we estimated a length of 8 bits but, since we work in a distributed environment, we are not able to predict the exact depth of the tree structure. Thus the computed string may need to be extended or it can happen that we must stop at a parent portion when traveling towards it. However, it is not really important which length l is chosen by the sender of the message since at any time any crossed peer can extend[7] the destination string without affecting previous steps. Therefore we are sure that every data inserted in the network can be retrieved even with no global knowledge about the network (and implicit W-Grid structure). This location service example is just one of the possible data management applications implementable in W-Grid. In fact, it is possible to manage each kind of one-dimensional or multi-dimensional data by translating them into binary string with the use of hashing algorithms.

---

of the multiplications are **1010** and *0110* respectively. Then, by crossing bit by bit the two string we get the c where destination node location is stored ***1***0***0***1***1***1***0***0*.

[7] See [25] for details

3.7.2 Local Learning

Local Learning (LL) a new feature we introduced in order to improve routing efficiency. The term learning is quite explicit with regard of what we aim to. The idea is to exploit messages routing and allow crossed nodes to learn something about the network so that they can use this knowledge for future routings. Local is referred to the fact that what nodes learn regards only their direct neighbors.

In a wireless environment unicast is never actually unicast, in fact, whenever a node communicates with one of its neighbors the communication is overheard by all of them, what it happens is that only the recipient of the communication will listen it. In the same way, each routing request exchanged among couples of nodes are heard by their respective neighbors. Our idea is that overhearing neighbors do not simply ignore the informations heard but they process them instead, finding for help to the routing nodes. It may happen that a node apparently farthest from the destination is aware of a node that would shorten the path, by giving back this information to its neighbor that was routing a message through another node it is possible that at the next routing the helping node will be chosen, and the path will be shorten. Simulation results show that the network gains in routing performances under this conditions.

3.7.3 Real Distance

We also added the Real Distance (RD) feature to W-Grid. Whenever a node n_j gets a coordinate from node n_i the new coordinate will be one bit longer that the father one. However n_i might have already split and while this information is known by n_j that will know about all the coordinates of it the same is not for n_j's neighbors which are not neighbors of n_i. Actually those neighbor could find useful such kind of information in order to get more precise distance values during routing. For this reasons routing table entry will also contain this integer value which represent the real distance among couple of nodes. In Chapter 4 we evaluated network performance with respect to this feature.

3.8 Nodes Failure

In ad-hoc networks nodes usually have scarce resource and they especially suffer of power constraints. This can lead to nodes failures that could affect routing efficiency. In W-Grid some robustness is guaranteed by multiple coordinates at each peer and by the adopted routing metric. In fact, it is

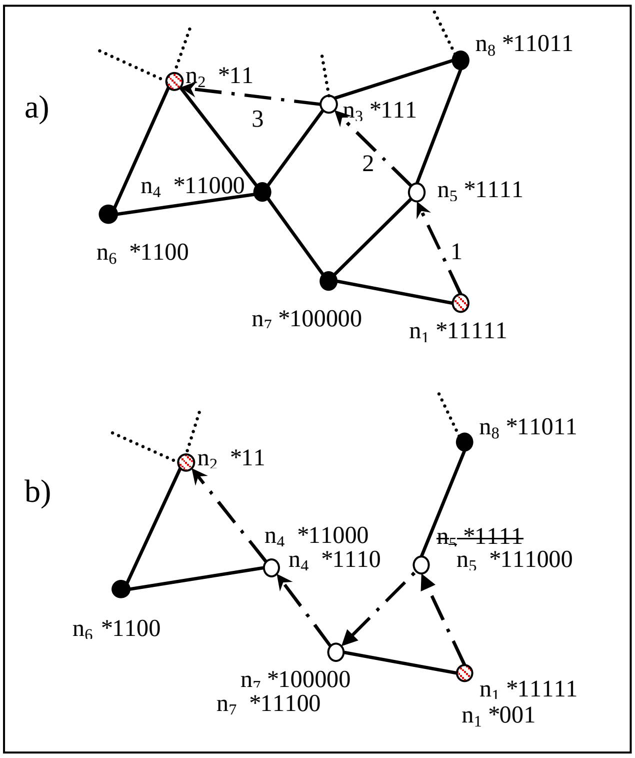


Figure 3.4: Effects of node failure (n_3) during routing of a packet from node n_1 to n_2

possible to route through different paths. If a broken path is discovered the packet can change direction (e.g. next hop) and follow a different path, according to another coordinate. However it may happen that a path breaks due to a node failure and no alternative way can be chosen.

In Figure 3.4 we present the case of a packet that must be routed to coordinate $*11$. During the routing a dead-end occurs, node n_5 cannot find any neighbor that improves its distance from the destination. This means that a link has broken since W-Grid total order relation guarantees the delivery in any case. When this happens the node deletes the coordinate that caused the dead-end and performs a "local broadcast" searching for the parent of the missing coordinate ($*11$ in our example). We use the term "local broadcast" since it is very likely that the searched coordinate will be close to the broadcasting node since it is a close relative of it. This means that the broadcast packet time-to-live will be small and its effects on network traffic will be limited. Once the coordinate has been found, the holding node fixes the relationship with the affected node by giving it a new coordinate, in our case through n_4 and n_7. It is important to specify that every recovery operation is lazy and triggered only on routing failures, in order to avoid any network efficiency loss.

3.8.1 Lazy recovery

In W-Grid we added a lazy recovery feature. In fact, besides active recovery we let the network try to fix situations not solved through the traffic normally generated by queries. Lazy recovery act as follows: whenever a node cannot recovery it gets in a recovery failed state. When a node is in this state it will first of all notice all its neighbors about it and its neighbors will do the same. Then, each node informed about this temporary state will add all of its coordinate to every query that it will be asked to route and that is evaluated to cross the node[8] in recovery failed state. The node in recovery failed state will scan each attached coordinate in the query message looking for a coordinate which is parent of the broken one, so that it can perform a recovery.

3.9 W*-Grid: Node Dependencies and Failure Recovery

The scope of W*-Grid extension is to guarantee network robustness to nodes (in particular sensors) or link failures while reducing network traffic and energy consumption. In W-Grid each single node failure cause all the direct children of the dead node/link to send a broadcast message searching for their grandfather (namely the father of the dead node), or for their closest ancestor, in order to find an alternative path to it and to place aliases (e.g. new coordinates) to be used for future routings directed towards the broken links. Although this solution works well it is quite expensive since, in order to be sure of finding the searched node, it is necessary to propagate the message several times, causing a high network traffic and overhead.

For this reason, taking inspiration from Menger theorem [16] we introduced a novel approach for generating the coordinate which builds several independent paths between a device d_i and its ancestors and viceversa. This means, for instance, that each node and its grandfather are jointed by at least two paths which do not share any node so that if one of d_i's fathers becomes unreachable along one path then the routing can be performed by following a different path (see Figure 3.5).

---

[8] each node can estimate if the query is likely to cross the orphan node by comparing query destination and the node

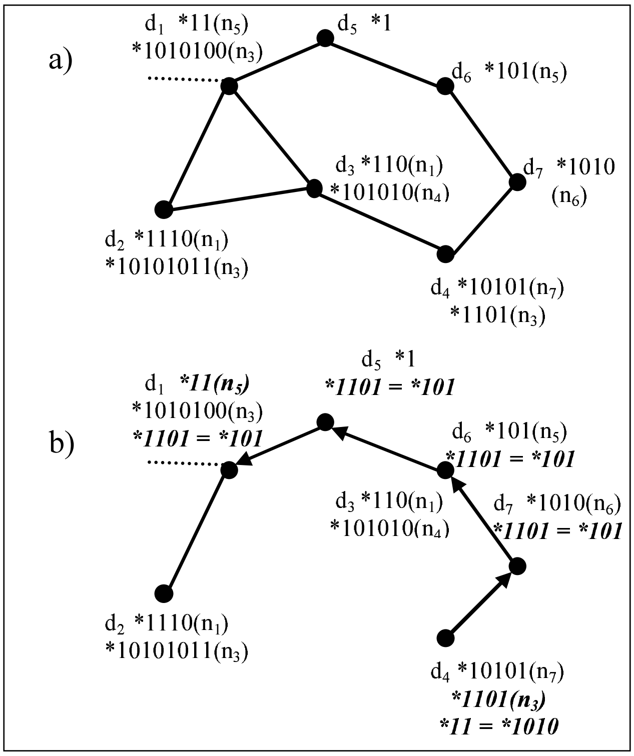


Figure 3.5: Grandfather discovery performed by node n_4 in case of father (node n_3) failure and aliases establishment

Given a walk w joining two devices d_1 and d_2 we define the sets of crossed node as:

$$CR(w) = \{d_i \in D : d_i \in w \wedge d_i \neq d_1 \wedge d_i \neq d_2\}$$

Two walks w_1 and w_2 from device d_1 to d_2 are independent if $CR(w_1) \cap CR(w_2) = \emptyset$. If we are able to create W*-Grid coordinates in a way that between a device d_i and its grandfather(s) there exist independent walks than we are able to guarantee that whenever a father of d_1 becomes unreachable another walk to d_1 grandfather that do not cross the unreachable father will exist. Walks independence is obtained by slightly changing the procedure that gives new coordinates to nodes.

In order to explain how W*-grid proceed we must first introduce the concept of nodes dependence. Given two devices d_1 and d_2 we say that d_2 depends on d_1 ($d_1 \rightarrow d_2$) if:

$$\forall c_{i1} \in d_1 \exists c_{j2} \in d_2 : father(c_{i1}) = c_{j2}$$

Namely, each coordinate in d_2 has been given by d_1 coordinates split. If $d_1 \rightarrow d_2$ in case of d_1 failure d_2 loses all the fathers of its coordinates, as a

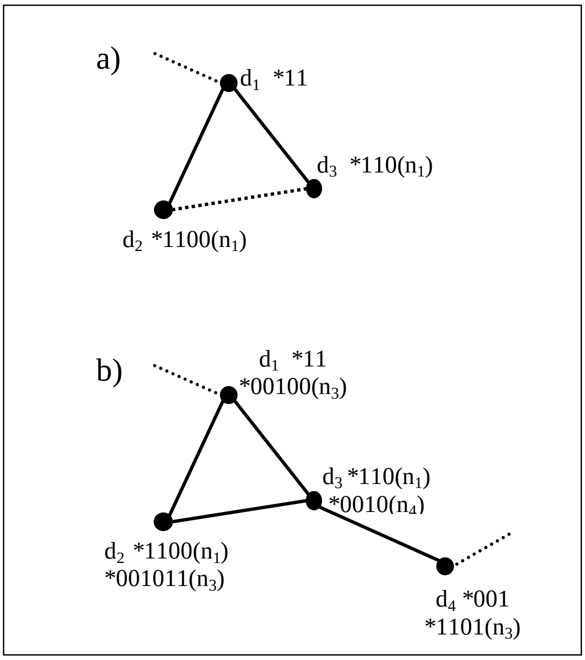


Figure 3.6: Coordinate generation with dependencies evaluation

consequence it will likely lose the links with its grandfather, making impossible to recovery the network from the failure. The situation in which a node depends on another nodes should therefore be avoided. The physical network may sometimes create situation of dependencies which are not avoidable, for instance, in Figure 3.6a) nodes d_2 and d_3 are dependent from d_1. Nodes independence is forced, when possible, by the following little expedient. When a device d_i need a coordinate from d_j it gathers d_j last received coordinates (namely the ones candidate to split) into a set we define $LAST_{d_j}$.

$$LAST(d_j) = \{last(d_i) : (d_j, d_i) \in L\}$$

Before choosing which coordinate in $LAST(d_j)$ will be split, d_i removes, from $LAST(d_j)$, all the coordinates that do not solve its dependencies with d_j or any other of its neighbors and that do not add independence to it. The coordinates c_i that must be taken out from $LAST$ are:

$$c_i : \exists c_j \in d_i, length(pref(c_i, c_j)) = length(c_j)$$

and:

$$c_i : \exists c_j \in NEIGH(i), length(pref(c_i, c_j)) = length(c_j)$$

Where $NEIGH(i)$ is the set of all the coordinates held by d_i neighbors, included d_j. If, due to dependence constraint $LAST = \emptyset$ than d_i does not take any coordinate.

In Figure 3.6b) it is shown the effect of this change in coordinates split. As we stated before, nodes d_2 and d_3 are both dependent from d_1, for this reason they do not get coordinates from each other until a new device d_4 joins the network and allow them to discover coordinates that make them independent from d_1. If d_2 and d_3 did not evaluate dependencies they would have exchanged coordinates likely reaching the limit in their coordinate number and preventing them to get other, more useful, coordinates in the future.

3.10 W^R-Grid: Replication in W-Grid

In this section we will focus on data replication, which is the contribution of W^R-Grid, an extension of W-Grid. In sensor networks the most important operations are data gathering and querying, therefore is necessary to guarantee the best efficiency during these tasks. In particular, data sensed by the network should be always available for users' queries and query execution latency must be minimized. In order to achieve these results we introduced replication of data in W^R-Grid. Data replication is obtained by generating multiple virtual coordinate spaces (namely multiple trees T). In this way, each information is replicated on every existing space, resulting in more than one benefit for network performances:

- **higher resistance to sensors failure**. Having multiple virtual spaces implies the existence of different paths for each coordinate and the possibility of changing routing space in case of dead-end;
- **reduction of query path length and latency**. Multiple realities mean multiple order relationship and therefore a reduction of the probability that two nodes physically close have very different virtual coordinates. Which may happen whenever a multi-dimensional space is translated into a one-dimensional space.

For what concerns replication implementation in W^R-Grid, we must say that the changes to the algorithm are few. Supposing that each sensor is given an unique identifier $ID(s)$, each reality is uniquely identified by the root node ID. Each coordinate c is coupled with its reality identifier so that each couple (ID, c) will be unique. During coordinate creation, sensors take a coordinate from every reality they discover from neighbors. At periodic beaconing, if any new reality is discovered a new coordinate from that reality is taken, allowing a progressive spread of the various realities to every participant of the network. During routing toward a target coordinate, sensors will evaluate their distance with respect to each reality and will route on the reality that takes closer to the target. Nothing else changes from what described in Chapter 3.

It is well known, from database literature, that replication has also drawbacks. Generally it has a negative impact in case of data updates, since it needs each existing replica to be affected by changes in order to maintain consistency. However we can observe that usually sensor networks are more like a stream of information in which older surveys can be replaced by newer ones or just stored with the newer one to maintain historical information. We can say that updates represent a limited problem and we can therefore focus on new data insertion. Since it is costly (in terms of network traffic) to replicate each tuple/record in each reality, analysis will be presented in Chapter 4 in order to find out the best replication configuration which guarantees query efficiency at reasonable costs.

Chapter 4

Experimental results

In order to evaluate the performances of W-Grid algorithm we implemented a Network Simulator in Java. We simulated network deployment upon areas having different dimensions and with various nodes densities (obtained by adjusting nodes transmission range). Nodes were randomly generated in but avoiding partitions in the network.

We let nodes to perform periodic beaconing. The beaconing is asynchronous, namely each peer is independent from the others, as it happens in real networks. Coordinate creation is gradual, the simulation randomly choose one node that beacons first and elects itself as root of a new virtual coordinate space. Then, as described in Chapter 3 we let that periodic beaconing builds the W-Grid network.

4.1 Average Path Length Comparisons

Simulation set 1. Nodes perform periodic beaconing (every 300ms) and generate messages at a parameterizable frequency. The beaconing is asynchronous, namely each peer is independent from the others, as it happens in real networks and we supposed a radio transmission range of 100 meters. Coordinate creation is gradual, the simulation randomly choose one node that beacons first and elects itself as root of a new virtual coordinate space. Then, as described in Chapter 3 we let that periodic beaconing builds the W-Grid network.

Once that every node had got its virtual coordinates the simulator generated 50000 messages between randomly chosen couples of sender/recipient nodes. Each message was routed according to our algorithm, following the virtual coordinates, and at the same time it was routed using GPSR algorithm (exploiting [x,y] physical positions of nodes). Obviously the com-

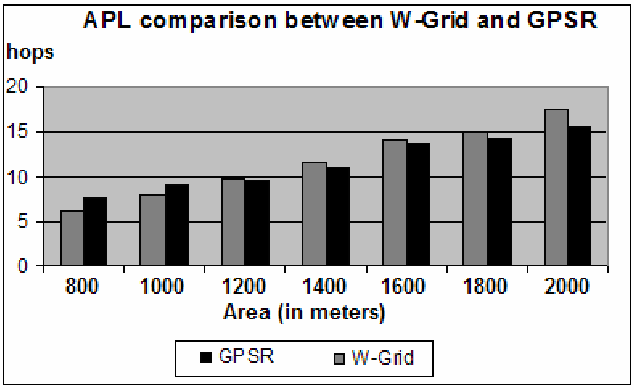


Figure 4.1: Average path length comparison between W-Grid and GPSR

	APL(in hops)		RMSE		Lost messages	
Area(nodes number)	WG	GPSR	WG	GPSR	WG	GPSR
800×800(120)	6,13	7,49	3,11	8,44	-	2,77%
1000×1000(200)	8,05	9,02	4,45	13,00	-	2,26%
1200×1200(290)	9,75	9,64	4,47	12,74	-	2,01%
1400×1400(400)	11,54	10,87	4,99	14,52	-	3,59%
1600×1600(520)	13,96	13,71	5,86	14,99	-	4,52%
1800×1800(660)	14,81	14,14	6,41	12,15	-	7,88%
2000×2000(820)	17,43	16,57	8,44	13,20	-	8,47%

Table 4.1: Results for different area dimensions (50 simulations each; 50000 messages sent)

parison is prohibitive, since GPSR can stay very close to the ideal routing algorithm also because it uses physical position of nodes. But our intention was to prove that W-Grid can return good performances anyway, especially considering that it doesn't require any kind of information about geographic position of nodes. This means not only a vaster and heterogeneous space of application, not limited only by GPS (or any other position estimation equipment) embedded devices, but also an easier deployment in every condition and everywhere. However, W-Grid returned amazing performances, especially considering that it doesn't require any kind of information about geographic position of nodes. This means not only a vaster and heterogeneous space of application, not limited only by GPS (or any other position estimation equipment) embedded devices, but also an easier deployment in every condition and everywhere. Figure 4.1 and Table 4.1 show that the

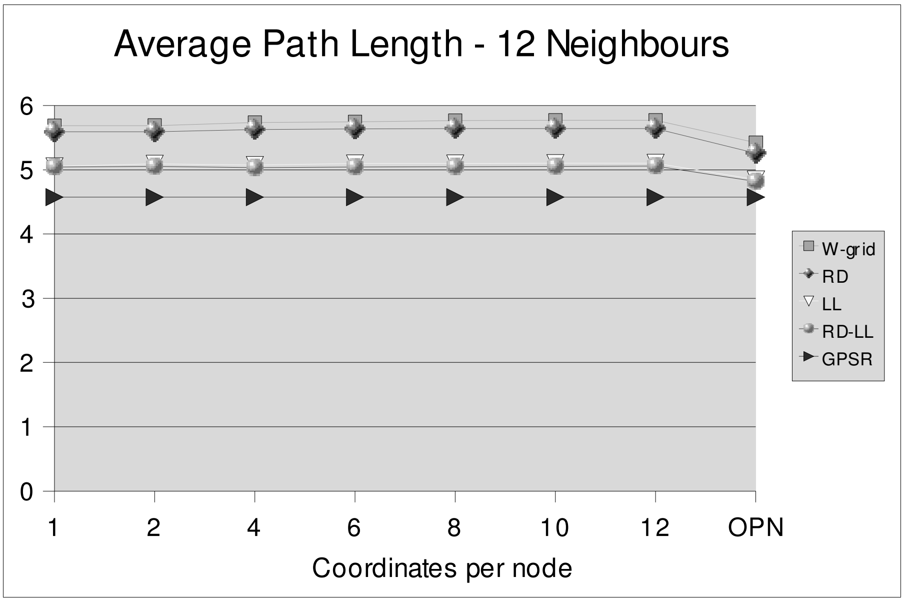


Figure 4.2: Query APL in a network with an average of 8 neighbors per node.

number of hops (APL) is almost equal in W-Grid and GPSR, but if we consider the natural advantage of GPSR that knows physical positions of the nodes we can say that the results are very good since, in some configurations our algorithm presents better performances, due to the perimeter issue of GPSR that may cause longest paths. Besides, it is important to say that W-Grid doesn't fail any message delivery and it performances are almost the same in the different runs per area showing that it is not affected by network topology. On the other side GPSR presents a notable percentage of routing failures and its performances are variable and dependent from nodes positions.

Simulation set 2. The simulation model consists of a square area $800 \times 800m$, in this area 205 nodes are randomly spread. Each node has its own ID and a radio range varying from $73m$ to $123m$ (ideal transmission) in order to get different densities, namely $4, 8$ and 12 neighbors per node respectively. For each scenario we ran 5 simulations and in each simulation we submitted 20000 queries to the system and then tested network robustness by turning off each node of the network one at a time. The simulator performed the following tasks:

- Random placement of nodes in a user-defined area;
- Generation of W-Grid coordinates at node exploiting implicit overhearing;

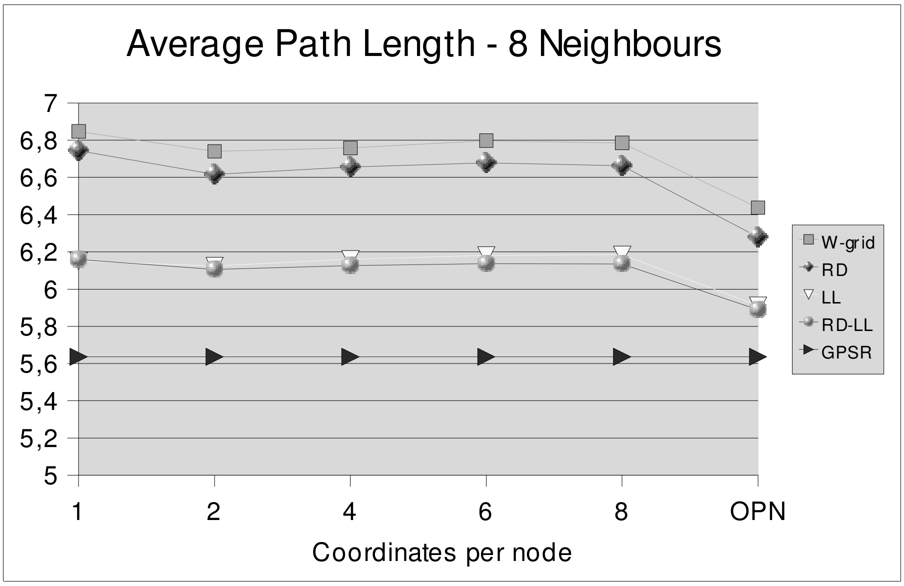


Figure 4.3: Query APL in a network with an average of 8 neighbors per node.

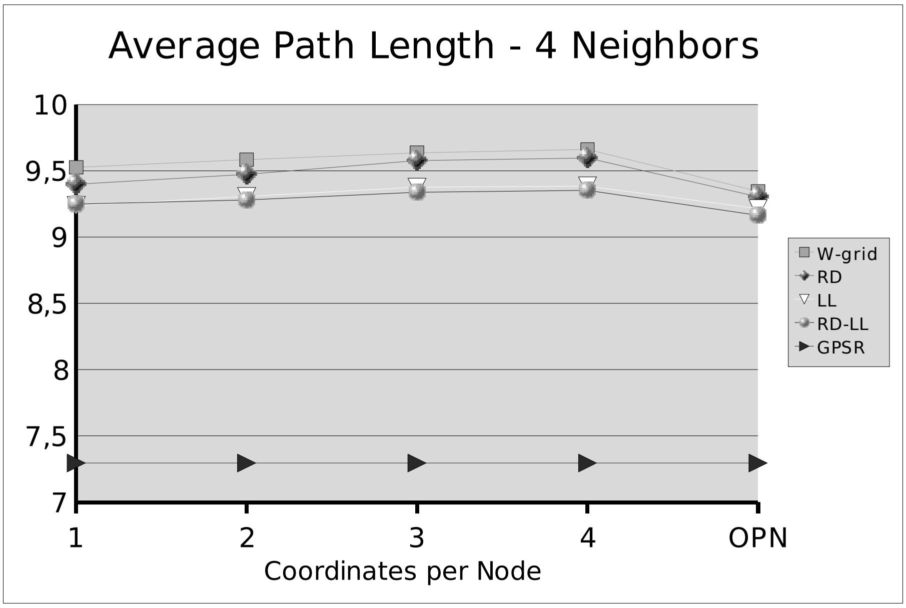


Figure 4.4: Query APL in a network with an average of 4 neighbors per node.

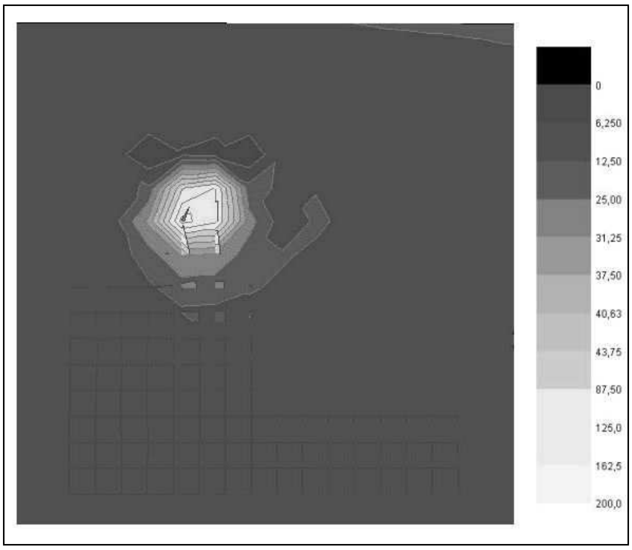


Figure 4.5: Contour showing storage load at nodes when SLOB algorithm is not running

- Random generation of 20000 queries;

For each simulation run we observed the variation in queries APL, namely the number of hops necessary to resolve a query, between W-Grid with Local Learning (LL) and Real Distance (RD) and GPSR.

Figures 4.2, 4.3 and 4.4 show that the number of hops (APL) is similar in W-Grid and GPSR especially when LL is applied. Besides, the flat look of the averages with respect with the number of coordinates shows that W-Grid behavior is stable according to that variable.

4.2 Load Balancing evaluation

The second aspect we focused on was load balancing at nodes in terms of data managed. Observing our implementation of location service we ran different simulation with and without using our SLOB algorithm. From Figures 4.5 and 4.6 we can see that its impact is really positive on storage load distribution among nodes. We used a bucket size $b = 1$ so that the system aims to achieve a perfect storage load balance with each peer that hold exactly one data. We can clearly see that in simulations where the algorithm is not used the percentage of nodes that store at least one data is less than 10%. Each node of this 10% manages on average 15,04 data and the root mean square error is 24,44. The situation is really unbalanced and

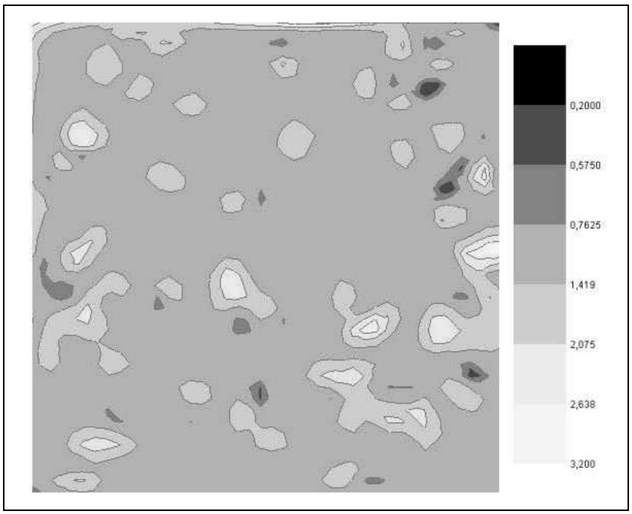


Figure 4.6: Contour showing storage load at nodes when SLOB algorithm is running

the most loaded node can have up to 200 data in worst cases. On the other side, by applying the algorithm we can take up to 90% (about 500 nodes out of 560) the number of nodes that store at least one data. In this case nodes manage about 1,14 data each and the root mean square error is 0,36.

4.3 Effects of Replication

We ran our Java simulator in order to evaluate the impact of multiple realities policy. We ran simulation on an area of 1500 by 1500 meters in which about 200 nodes with a supposed radio transmission of 100 meters were spread. Coordinate creation is gradual, the simulator randomly choose one or more nodes to elect as root of realities, then, as described in Chapter 3 we let periodic beaconing to build the W^R-Grid network. Beside coordinate creation we simulated the survey of events (3000 in each run) by nodes and their consequent insertion in the network.

We also simulated the execution of queries of randomly chosen data from randomly chosen nodes. Simulation reported information about the number of hops covered by queries (query path length), the number of data stored per node (storage load) and the number of times each node is request to route a query (workload) during the simulation. We analyzed average and Mean Square Error of those measures with different numbers of replicas in the system and different query/insertion ratios (10/1, 5/1). Figure 4.7 shows

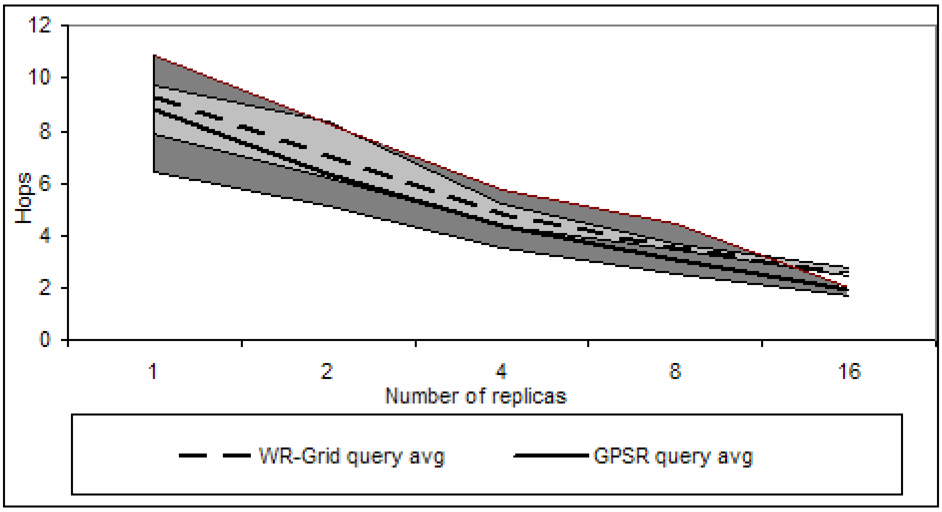
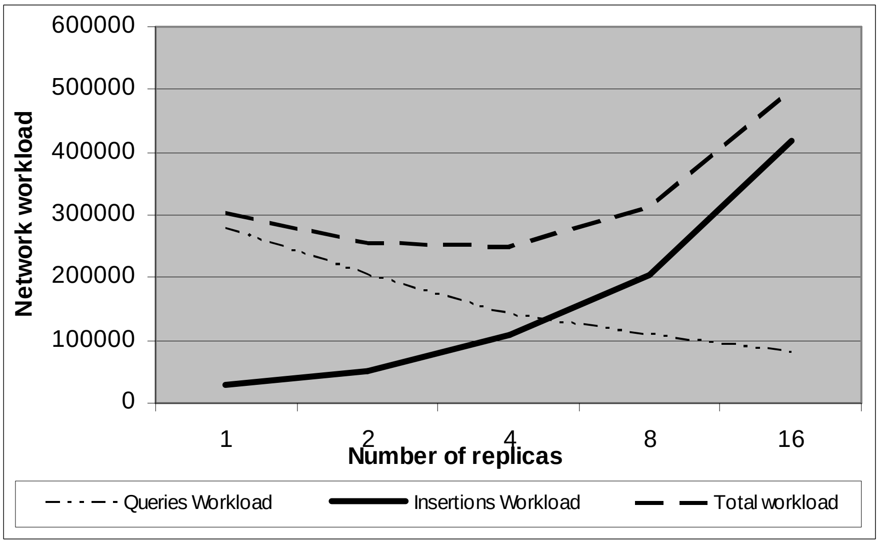


Figure 4.7: APL for different numbers of realities in the network

that as the number of realities increases the routing performances of W^R-Grid improves considerably (average hops are halved compared to W-Grid). This is the demonstration that multiple realities reduce the probability that two nodes physically close are distant according to the order relationship. It is important to notice that this benefit follows a logarithmic curve, therefore, once that a certain number of coordinate (we can say around 10) is reached, it is no more convenient to increase it.

In Figure 4.8 and 4.9 can be observed a consequence of the improvement in routing efficiency. Since the average hops per query is reduced also the average node workload is reduced. At the same time it is possible to see that the MSE of that measure decreases, meaning a better balance in the workload per node. By observing Figure 4.9 we can say that multiple realities improve storage load balancing too and surely this has a positive effect on nodes energy consumption since it implies a more balanced request load per node.

On the other side replication implies higher cost at insertion time, more precisely, in case of n realities each event must be inserted in n different indexes. Therefore the number of replica should be limited to the smallest necessary in order to guarantee data availability and routing efficiency. From our simulations and showed graphs we can say that a number of 4-6 realities is the best choice. With a higher number the increase of routing efficiency and balancing cannot be justified by the increase of replication costs.



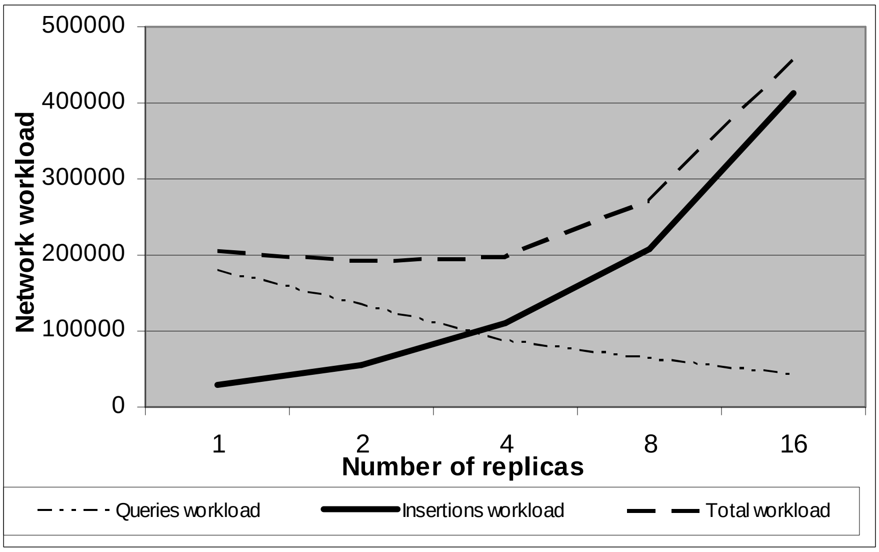


Figure 4.8: Sensors workload for different numbers of replicas and different Query/insertion ratio (10/1 and 5/1)

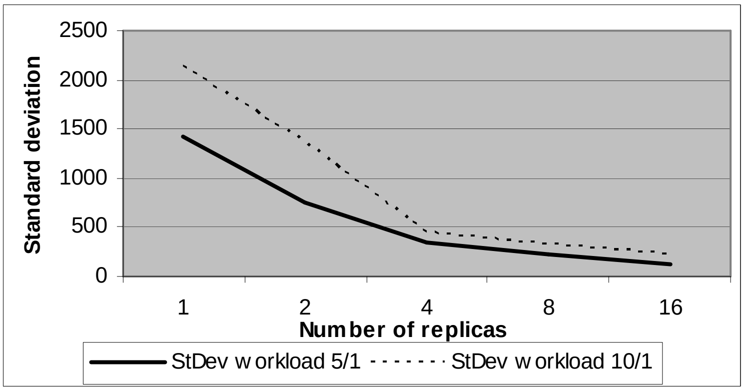


Figure 4.9: MSE of nodes workload for different numbers of replicas.

4.4 Recovery failures

The simulation model consists of a square area 800 × 800m, in this area 205 nodes are randomly spread. Each node has its own ID and a radio range varying from 73m to 123m (ideal transmission) in order to get different densities, namely $4, 8$ and 12 neighbors per node respectively. For each scenario we ran 5 simulations and in each simulation we submitted 20000 queries to the system and then tested network robustness by turning off each node of the network one at a time. The simulator performed the following tasks:

- Random placement of nodes in a user-defined area;
- Generation of W-Grid coordinates at node exploiting implicit overhearing;
- Random generation of 20000 queries;
- Turning off of nodes at the delivery of queries, as previously described.

For each simulation run we observed:

- The variation in queries APL (Average Path Length), namely the number of hops necessary to resolve a query, between W-Grid with LL and RD and GPSR.
- The ratio of succeeded recovery in W-Grid scenarios;

In order to show W*-Grid robustness in case of single node failure while saving energy by avoiding message broadcast to recovery from failure we run another set of simulations. We gathered results regarding the network routing performances, in term of average path length, and robustness.

The second measure we evaluate is the ratio of failure recovery which is correctly performed according to the different node densities and the number of coordinates. We simulate two different recovery strategies:

- Active recovery;
- Lazy recovery.

Lazy recovery is performed whenever a node could not solve a failure situation with the active recovery. We present the results obtained with both strategies.

In Figures 4.10, 4.11 and 4.12 five curves are represented. We basically compare W-Grid efficiency with coordinates dependencies against the W-Grid solution exploiting message broadcast. The broadcast has been tried with different TTLs and obviously its performances improve as TTL increases. The fifth curve represents an unlimited broadcast which has been simulated whenever W-Grid could not be able to perform recovery. Figures show that almost every time that W-Grid was not able to perform recovery, unlimited broadcast was not able as well, meaning that W-Grid failed just because the network was partitioned due to device failure. Figures 4.13, 4.14 and 4.15 show the lazy recovery procedure that exploits routing of queries to discover lost relatives. Simulations returned that lazy recovery might help

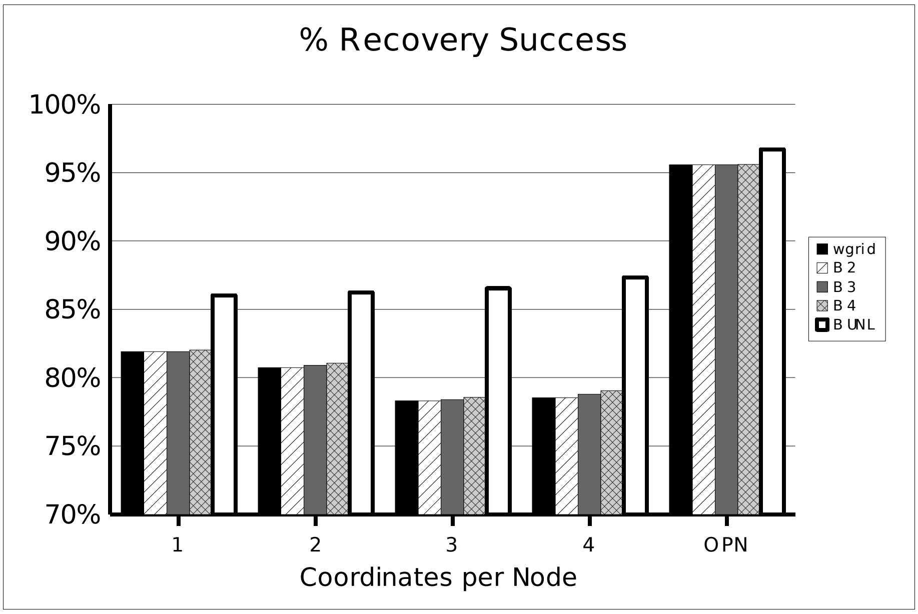


Figure 4.10: Recovery success ratio with an average of 4 neighbors per node.

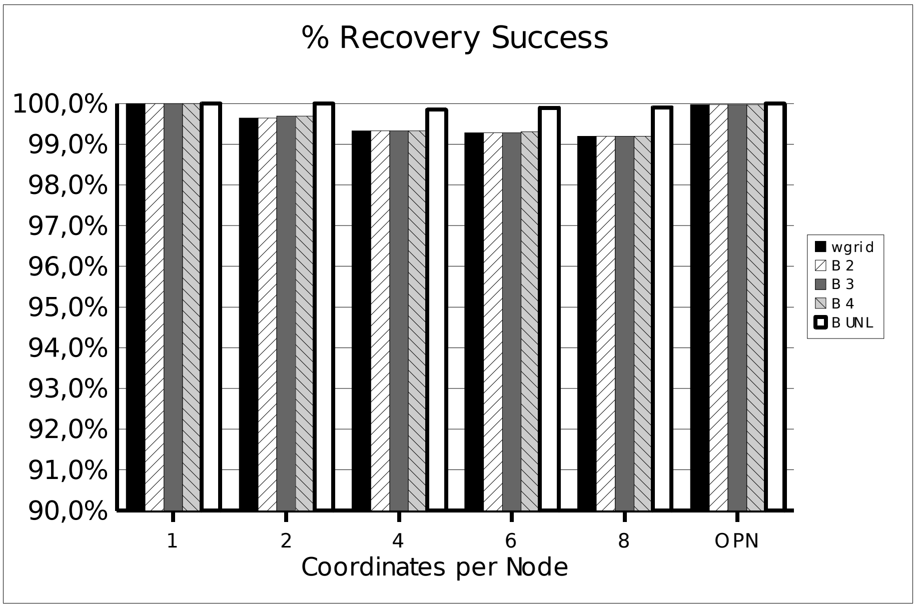


Figure 4.11: Recovery success ratio with an average of 8 neighbors per node.

W-Grid to get closer to the unlimited broadcast performances. In Figure 4.16 we can see that the percentage of successfull query keeps really high even when the network is in an instable state due to recoveries failed. Figures 4.17, 4.18 and 4.19 the network traffic generated by W-Grid active failure recovery strategies compared with broadcast applicated to W-Grid (with different level of broadcast propagations). Figures show that W-Grid heavily reduces the number of messages required for recovery. We don't show the cost required by lazy recovery since it actually doesn't add any message in the network. Please remember that lazy recovery exploits messages that traverse the network due to queries. Lazy recovery require some nodes to inspect messages for a certain time interval. These inspection, however, require insignificant amount of time with compare to the transmission latency.

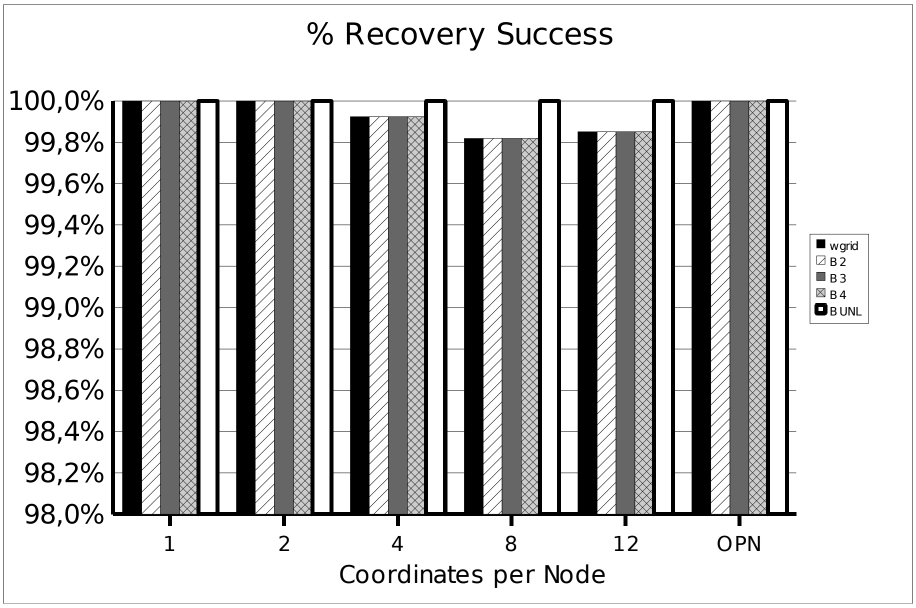


Figure 4.12: Recovery success ratio with an average of 12 neighbors per node.

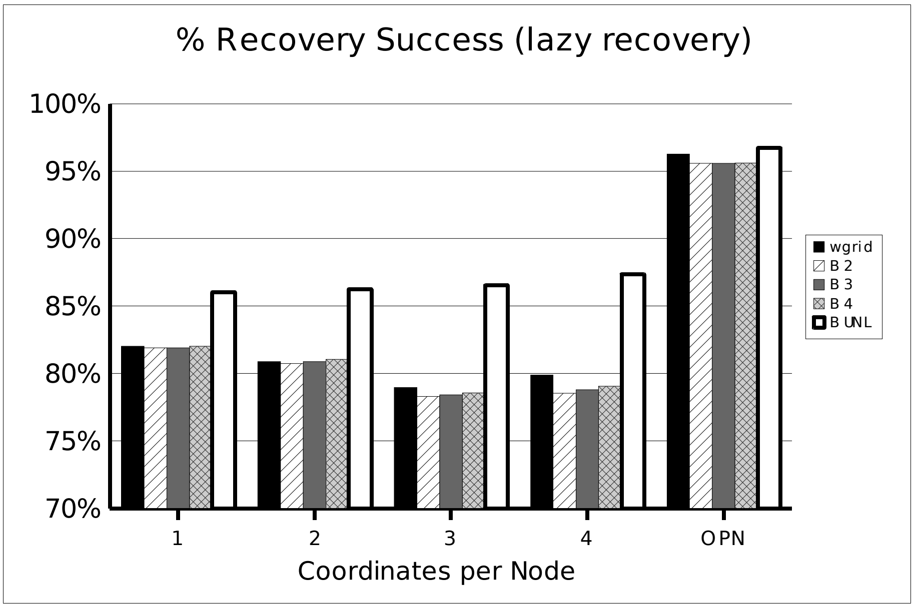


Figure 4.13: Recovery failure ratio with an average of 4 neighbors per node after lazy recovery.

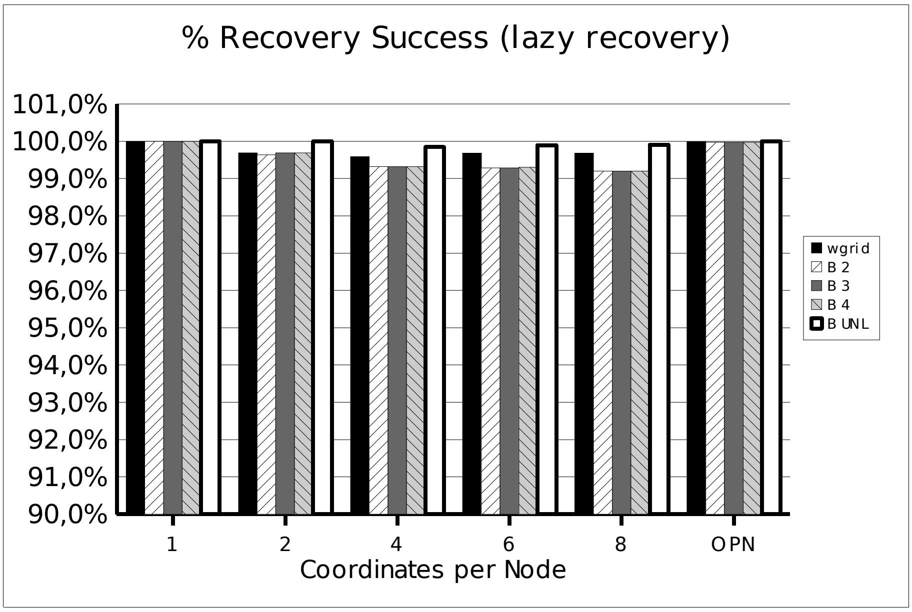


Figure 4.14: Recovery failure with an average of 8 neighbors per node after lazy recovery.

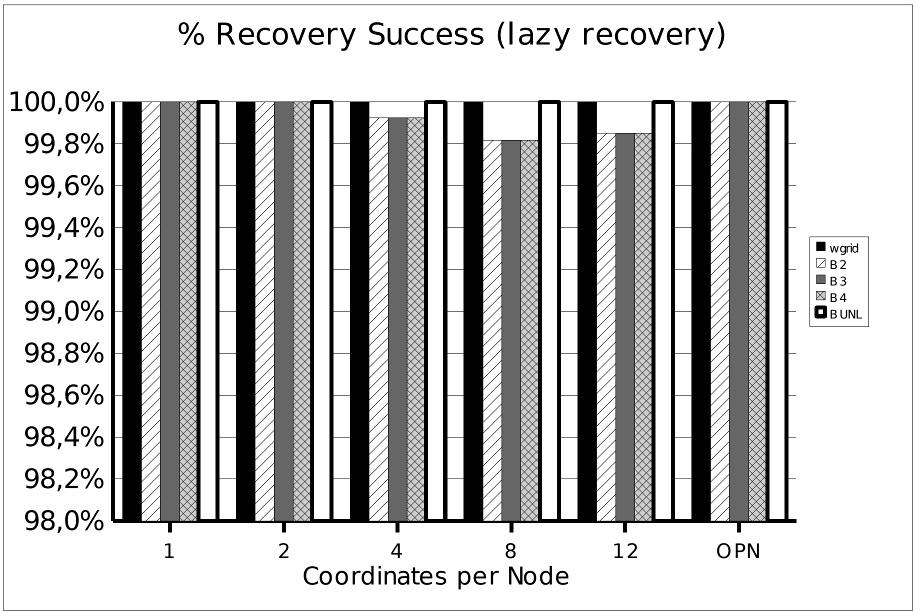


Figure 4.15: Recovery failure ratio with an average of 12 neighbors per node after lazy recovery.

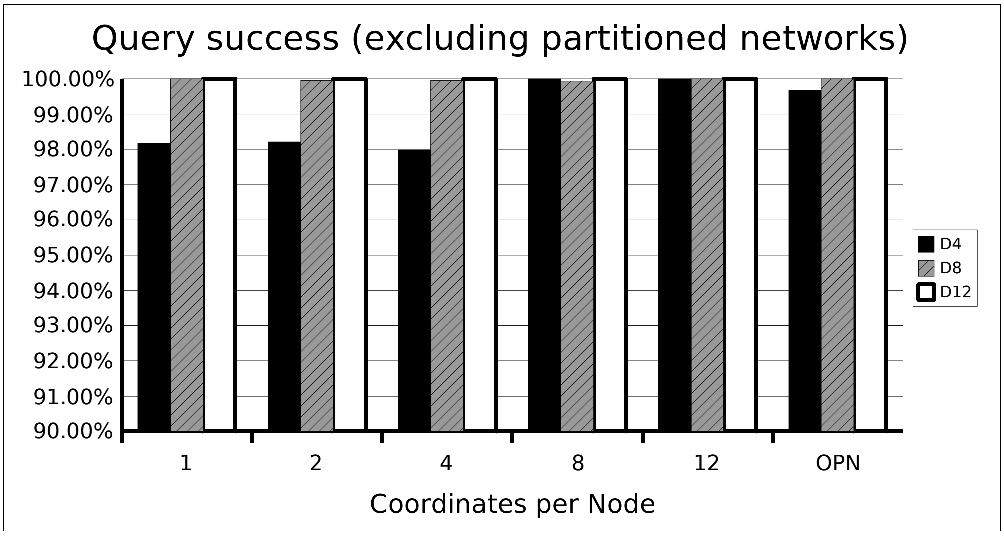


Figure 4.16: Percentage of successfull query in case of recovery failure.

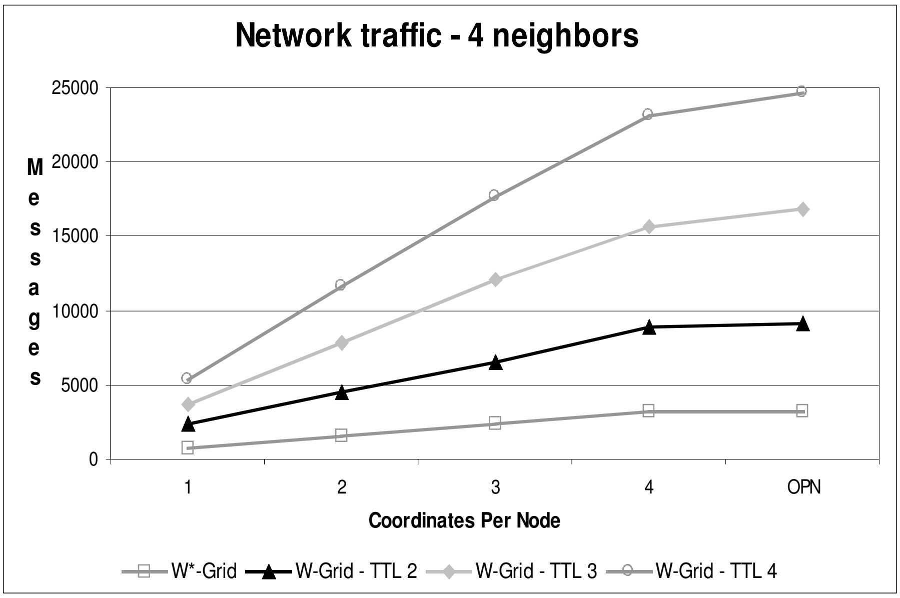


Figure 4.17: Network traffic generated by recovery in a network with an average of 4 neighbors per node.

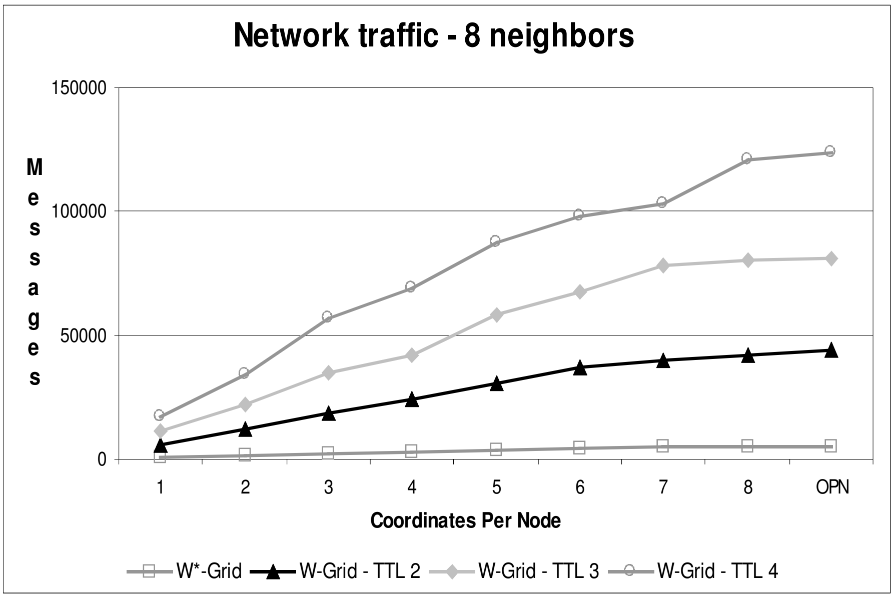


Figure 4.18: Network traffic generated by recovery in a network with an average of 8 neighbors per node.

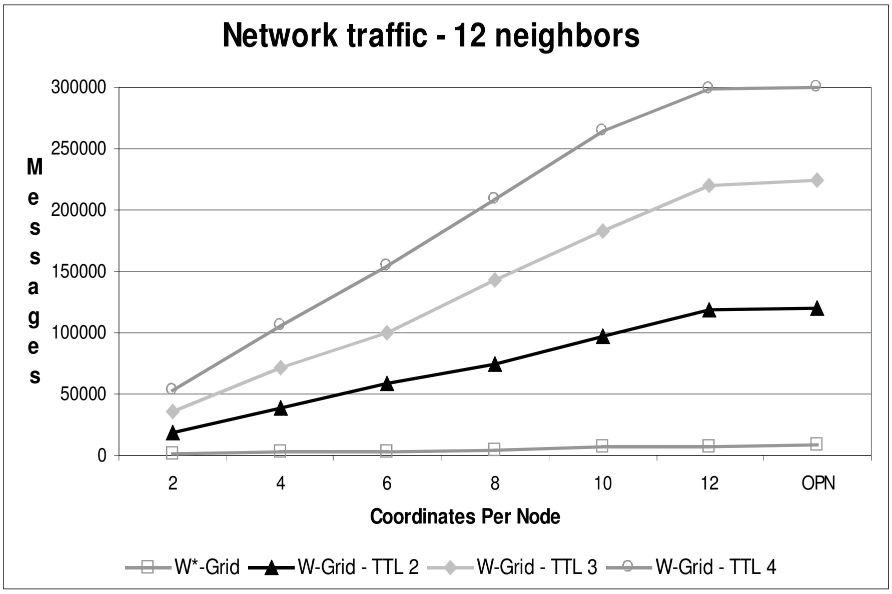


Figure 4.19: Network traffic generated by recovery in a network with an average of 12 neighbors per node.

Chapter 5

Conclusions

We studied W-Grid, a novel decentralized infrastructure that self-organizes wireless devices in an ad-hoc network, where each node has one or more virtual coordinates through which both message routing and data management occur without reliance on either flooding/broadcasting operations or GPS. The resulting network does not suffer from the dead-end problem, which happens in geographic-based routing when a node is unable to locate a neighbor closer to the destination than itself.

We extended W-Grid to make it a fault tolerant cross-layer infrastructure W*-Grid for routing and multi-dimensional data management in ad-hoc sensor networks. We explained the modifications in the model thanks to which W*-Grid recovers from nodes and/or connectivity failures without using broadcasting/multi-cast transmissions. The main contribution of W*-Grid is that, in case of failures, the resulting wireless multi-hop networks drastically reduce the energy consumption while guaranteeing robustness and preserving the same W-Grid performance and properties. This result has been achieved by defining a novel decentralized technique for the assignment of coordinates, according to which, by construction, between each node and its ancestors (and vice versa) exists at least two disjoint paths, namely two paths which do not share any node. We also worked on ^{R}-Grid which extends W-Grid by adopting a replication methodology. W^R-Grid acts as a distributed database without needing neither special implementation nor reorganization and any kind of data can be distributed, stored and managed.

An extensive number of simulations showed significant performance when compared with GPSR and performance measures of W*-Grid remain unchanged, such as the average path length under several device densities, or get better, such as the reduction of both network traffic and the total number of coordinates in the system. We have also evaluated the benefits of replication on data management with W^R*-Grid, discovering from experimental result that it can halve the average number of hops in the network. The direct consequence of these results are a significant improvement on energy consumption and a workload balancing among sensors (number of messages routed by each node). Finally, thanks to the replications, whose number can be arbitrarily chosen, the resulting sensor network tolerates sensors disconnections/connections due to failures of sensors.

Appendix A

W-Grid Simulator Code

Interface WGActor

```
package it.unibo.deis.gmonti.netsimulator.wgrid;

import java.util.ArrayList;

import it.unibo.deis.gmonti.netsimulator.mobilenode.
    MobileNodeActor;

public interface WGActor extends MobileNodeActor {

   public String getWGBinaryId();
   public ArrayList<WGReality> getWGRealities();
   public int getWGNodeMaxLengthVC();
   public double getWGPacketsWalkedDistance();
   public int getWGReceivedPackets();

   //advanced get methods
   public WGReality getWGReality(int reality);
   public WGReality getWGReality(WGReality reality);
   public int getWGRealitiesSize();
   public int getWGCoordinatesSize(int reality);
   public ArrayList<? extends WGActor> getNeighbors();
   public boolean hasNeighbor(int id);

   //set methods
   public void addToWGRealities(WGReality r);
   public void checkWGNodeMaxLengthVC();
```

```
    public void addToWGPacketsWalkedDistance(double d);
    public void incWGReceivedPackets();

    //recovery methods
    public boolean recoveryFailed();
}
```

Class WGReality

```
package it.unibo.deis.gmonti.netsimulator.wgrid;

import java.util.ArrayList;

public class WGReality {
    private int rootId;
    private ArrayList<WGCoordinate> coordinates;
    private ArrayList<WGCoordinate> givableCoordinates;
    private int maxLengthVC;
    private ArrayList<WGRoutingTableEntry> routingTable;
         //Routing table: list of
         WGridRoutingTableEntries
    public WGActor mostLoadedNeighbor;
    public int mostLoadedNeighborLoad;
    public int mostLoadedNeighborNeighborLoad;

    private ArrayList<Shortener> locals; //Local
         Learning (LL)

    public WGReality(int root){
        rootId = root;
        coordinates = new ArrayList<WGCoordinate>();
        givableCoordinates = new ArrayList<WGCoordinate
             >();
        maxLengthVC = 0;
        routingTable = new ArrayList<WGRoutingTableEntry
             >();
        locals = new ArrayList<Shortener>();
        mostLoadedNeighbor = null;
        mostLoadedNeighborLoad = 0;
        mostLoadedNeighborNeighborLoad = 0;
    }

    //Get methods
    public int getRootId() {
        return rootId;
    }
    public ArrayList<WGCoordinate> getCoordinates(){
        ArrayList<WGCoordinate> result = new ArrayList<
```

```
            WGCoordinate>();
        for (WGCoordinate c : coordinates)
            result.add(c);
        return result;
    }
    public ArrayList<WGCoordinate>
        getNotSplitCoordinates(){
        ArrayList<WGCoordinate> result = new ArrayList<
            WGCoordinate>();
        for (WGCoordinate c : coordinates)
            if (!c.hasSplit())
                result.add(c);
        return result;
    }
    public ArrayList<WGCoordinate> getSplitCoordinates()
        {
        ArrayList<WGCoordinate> result = new ArrayList<
            WGCoordinate>();
        for (WGCoordinate c : coordinates)
            if (c.hasSplit())
                result.add(c);
        return result;
    }
    public ArrayList<WGCoordinate>
        getAllNotSplitCoordinates(){
        ArrayList<WGCoordinate> result = new ArrayList<
            WGCoordinate>();
        for (WGCoordinate c : coordinates)
            if (!c.hasSplit())
                result.add(c);
        for (WGCoordinate c : givableCoordinates)
            result.add(c);
        return result;
    }
    public ArrayList<WGCoordinate>
        getNotEmptyCoordinates(){
        ArrayList<WGCoordinate> result = new ArrayList<
            WGCoordinate>();
        for (WGCoordinate c : coordinates)
            if (!c.hasSplit() && c.getManagingDataSize()
                >0)
```

```
            result.add(c);
      for (WGCoordinate c : givableCoordinates)
         if (c.getManagingDataSize() >0)
            result.add(c);
      return result;
   }
   public int getCoordinatesSize() {
      return getCoordinates().size();
   }
   public int getNotSplitCoordinatesSize() {
      return getNotSplitCoordinates().size();
   }
   public int getAllNotSplitCoordinatesSize() {
      return getAllNotSplitCoordinates().size();
   }
   public int getNotEmptyCoordinatesSize(){
      return getNotEmptyCoordinates().size();
   }
   public ArrayList<WGCoordinate> getGivableCoordinates(){
      ArrayList<WGCoordinate> result = new ArrayList<WGCoordinate>();
      for (WGCoordinate c : givableCoordinates)
         result.add(c);
      return result;
   }
   public ArrayList<WGCoordinate> getAllCoordinates(){
      ArrayList<WGCoordinate> result = new ArrayList<WGCoordinate>();
      for (WGCoordinate c : coordinates)
         result.add(c);
      for (WGCoordinate c : givableCoordinates)
         result.add(c);
      return result;
   }
   public WGActor getMostLoadedNeighbor(){
      return mostLoadedNeighbor;
   }
   public int getMostLoadedNeighborLoad(){
      return mostLoadedNeighborLoad;
   }
```

```
    public int getMostLoadedNeighborNeighborLoad(){
        return mostLoadedNeighborNeighborLoad;
    }
    public void setMostLoadedNeighbor(WGActor a){
        mostLoadedNeighbor = a;
    }
    public void setMostLoadedNeighborLoad(int l){
        mostLoadedNeighborLoad = l;
    }
    public void setMostLoadedNeighborNeighborLoad(int l){
        mostLoadedNeighborNeighborLoad = l;
    }
    public int getMaxLentghVC() {
        return maxLengthVC;
    }
    public ArrayList<WGRoutingTableEntry> getRoutingTable() {
        return routingTable;
    }
    public void checkMaxLengthVC() {
        for (WGCoordinate c : coordinates)
            if (c.getVC().length() > maxLengthVC)
                maxLengthVC = c.getVC().length();
        for (WGCoordinate c : givableCoordinates)
            if (c.getVC().length() > maxLengthVC)
                maxLengthVC = c.getVC().length();
        for (WGRoutingTableEntry rte : routingTable)
            if (rte.getVC().length() > maxLengthVC)
                maxLengthVC = rte.getVC().length();
    }
    public double getSpacePortion() {
        double result = 0d;
        ArrayList<WGCoordinate> nsc = getAllNotSplitCoordinates();
        for (WGCoordinate c : nsc) {
            result += 1d/Math.pow(2d, c.getVC().length()-1);
        }
        return result;
    }
```

```

  public WGCoordinate getCoordinateFromVC(String s) {
    for (WGCoordinate c : coordinates)
      if (c.getVC().equals(s))
        return c;
    for (WGCoordinate c :  givableCoordinates)
      if (c.getVC().equals(s))
        return c;
    return null;
  }

  public int getRoutingTableSize() {
    return routingTable.size();
  }//getRoutingTable
  public ArrayList<WGActor> getDistinctNeighbors(){
    ArrayList<WGActor> neighbors = new ArrayList<
         WGActor>();
    for (WGRoutingTableEntry rte : routingTable){
      boolean has = false;
      for (WGActor mgn : neighbors){
      if (mgn.getMobileNodeId() == rte.
           getReferredNodeId())
        has = true;
      }
      if (!has)
        neighbors.add(rte.getReferredNode());
    }
    return neighbors;
  }
  public ArrayList<WGRoutingTableEntry>
       getRoutingTableEntriesForNode(int nodeId){
    ArrayList<WGRoutingTableEntry> entries = new
         ArrayList<WGRoutingTableEntry>();
    for (WGRoutingTableEntry rte : routingTable){
      if (rte.getReferredNodeId() != nodeId)
      continue;
      entries.add(rte);
    }
    return entries;
  }
  public WGRoutingTableEntry getRoutingTableEntry(
```

```
        WGActor n, WGCoordinate c) {
        for (WGRoutingTableEntry rt : routingTable){
        if (rt.getReferredNodeId() == n.getMobileNodeId()
            && c.getVC() == rt.getVC())
            return rt;
        }
        return null;
    }
    public ArrayList<WGActor> checkIfHasOnePerNeighbor(
        WGActor no){
        ArrayList<WGActor> missing = new ArrayList<
            WGActor>();
            for (WGActor n : getDistinctNeighbors()){
            if (n.getWGRealitiesSize() == 0)
                continue;
            boolean has = false;
            for (WGCoordinate c : coordinates){
                if (c.getNodeFatherId() == n.
                    getMobileNodeId())
                has = true;
            }
            if (!has)
                missing.add(n);
        }
        return missing;
    }
    public int coordinatesPerNeighbor(int n){
        int res = 0;
        for (WGCoordinate c : coordinates){
            if (c.hasSplit())
                continue;
            if (c.getNodeFatherId() == n)
                res++;
        }
        return res;
    }

    //set methods
    public void addCoordinate(WGCoordinate c) {
        coordinates.add(c);
        c.makeNotGivable();
```

```
    }
    public void addGivableCoordinate(WGCoordinate c) {
        givableCoordinates.add(c);
        c.makeGivable();
    }
    public boolean setSplit(WGCoordinate c, WGActor
        askingNode){
        c.setSplit(askingNode);
        if (c.isGivable()) {
            c.makeNotGivable();
            coordinates.add(c);
            return givableCoordinates.remove(c);
        }
    return true;
    }
    public boolean moveCoordinate(WGActor n,
        WGCoordinate coord) {
        WGReality givingNodeReality = coord.getOwner().
            getWGReality(this);
        if (!givingNodeReality.givableCoordinates.remove(
            coord))
            return false;
        coord.setOwner(n);
        addCoordinate(coord);
        return true;
    }
    public void clearRoutingTable() {
        routingTable.clear();
    }
    public void addToRoutingTable(WGRoutingTableEntry
        rte) {
        routingTable.add(rte);
    }
    public boolean removeFromRoutingTable(int nodeId,
        String vc) {
        int i;
        for (i = 0; i < routingTable.size(); i++)
            if (routingTable.get(i).getReferredNodeId() ==
                    nodeId && routingTable.get(i).getVC().
                equals(vc))
            break;
```

```
        if (i == routingTable.size())
            return false;
        routingTable.remove(i);
        return true;
    }
    public void addShortener(WGActor n, String s){
        locals.add(new Shortener(n,s));
    }
    public String toString() {
        return "" + rootId;
    }
}
//Inner class Shortener
class Shortener{
    String coordinate;
    int nodeId;
    WGActor node;

    Shortener(WGActor n, String s){
        coordinate = s;
        node = n;
        nodeId = n.getMobileNodeId();
    }
}
```

Class WGCoordinate

```
package it.unibo.deis.gmonti.netsimulator.wgrid;

import java.util.ArrayList;

public class WGCoordinate {

   private int realityId;
   private String VC;
   private int nBit;
   private boolean nearing;
   private WGActor owner;
   private int nodeFatherId;
   private WGActor nodeFather;
   private boolean isGivable;
   private boolean hasSplit;
   private int childNodeId;
   private WGActor childNode;

   //sensor part
   private ArrayList<String> managingData;

   /**
    * @param VC the string rapresentation of the VC
    * @param rootID the father of the system
    */
   public WGCoordinate(int r, String c, WGActor ow,
        WGActor father){
      realityId = r;
      VC = c;
      nBit = 0;
      nearing = false;
      owner = ow;
      nodeFather = father;
      if (father != null)
         nodeFatherId = father.getMobileNodeId();
      else
         nodeFatherId = -1;
      isGivable = false;
      hasSplit = false;
```

```
        childNodeId = -1;
        childNode = null;
    }

    public WGCoordinate(int r, String c, WGActor ow, WGActor father, boolean isG){
        realityId = r;
        VC = c;
        nBit = 0;
        nearing = false;
        owner = ow;
        nodeFather = father;
        if (father != null)
            nodeFatherId = father.getMobileNodeId();
        else
            nodeFatherId = -1;
        isGivable = isG;
        hasSplit = false;
        childNodeId = -1;
        childNode = null;
    }
    public int getRealityId(){
        return realityId;
    }
    public String getVC(){
        return VC;
    }
    public int getNBit(){
        return nBit;
    }
    public void setNBit(int nb){
        nBit = nb;
    }
    public boolean getNearing() {
        return nearing;
    }
    public void setNearing(boolean sn) {
        nearing = sn;
    }
    public WGActor getOwner(){
        return owner;
```

```
    }
    public void setOwner(WGActor now){
        owner = now;
    }
    public WGActor getNodeFather(){
        return nodeFather;
    }
    public int getNodeFatherId() {
        return nodeFatherId;
    }
    public boolean isGivable() {
        return isGivable;
    }
    public void makeGivable() {
        isGivable = true;
    }
    public void makeNotGivable() {
        isGivable = false;
    }
    public boolean hasSplit() {
        return hasSplit;
    }
    public int getChildNodeId(){
        return childNodeId;
    }
    public WGActor getChildNode(){
        return childNode;
    }
    public void setSplit(WGActor child) {
        hasSplit = true;
        if (child != null){
            childNodeId = child.getMobileNodeId();
            childNode = child;
        }
        else{
            childNodeId = -1;
            childNode = null;
        }
    }
    public ArrayList<String> getManagingData(){
        return managingData;
```

```
    }
    public void addToManagingData(String s) {
        managingData.add(s);
    }
    public int getManagingDataSize(){
        return managingData.size();
    }
    public void removeFromManagingData(String s){
        managingData.remove(s);
    }
    public boolean hasWGData(String s){
        for (String t : managingData)
            if (t.equals(s))
                return true;
        return false;
    }
    public String maxCommonPrefix(WGCoordinate c) {
        String result = null;
        int i = 0;
        while (i < VC.length() && i < c.VC.length() ) {
            if (VC.charAt(i) == c.VC.charAt(i))
                result += VC.charAt(i);
            i++;
        }
        return result;
    }
    public int maxCommonPrefixLength(WGCoordinate c) {
        String result = null;
        int i = 0;
        while (i < VC.length() && i < c.VC.length() )
            if (VC.charAt(i) == c.VC.charAt(i)) {
                result += VC.charAt(i);
                i++;
            }
        return i;
    }
    public String maxCommonPrefix(String s) {
        String result = null;
        int i = 0;
        while (i < VC.length() && i < s.length() ) {
            if (VC.charAt(i) == s.charAt(i))
```

```
                result += VC.charAt(i);
            i++;
        }
        return result;
    }
    public int maxCommonPrefixLength(String s) {
        String result = null;
        int i = 0;
        while (i < VC.length() && i < s.length() )
            if (VC.charAt(i) == s.charAt(i)) {
                result += VC.charAt(i);
                i++;
            }
        return i;
    }
}
```

Class WGrid

```
package it.unibo.deis.gmonti.netsimulator.wgrid;

//import it.unibo.deis.gmonti.eventmanager.EM;
import it.unibo.deis.gmonti.em.EM;
import it.unibo.deis.gmonti.netsimulator.NS;
import it.unibo.deis.gmonti.netsimulator.NSParameters;
import it.unibo.deis.gmonti.netsimulator.nsactor.NSActorManager;

import java.util.ArrayList;
import java.util.Iterator;

public class WGrid {

    public static final int ONE_PER_NEIGHBOR = -1;

    //Periodic event needed to update routing table and in some case setting new coordinates
    public static boolean beaconing(WGActor actor, ArrayList<? extends WGActor> receivingNodes,
        int numberOfRealities, int coordsPerNode, int coordsPerBeacon, int coordsPerNeighbor, boolean checkDependencies,
        int SLOB, double avgThreshold, double sqmThreshold){
        boolean createdWGCoordinate = false;
        ArrayList<WGActor> tempNeighbors = new ArrayList<WGActor>();

        //physical beaconing
        for (WGActor receivingNode : receivingNodes){// For each node in the radio range, updates the routing table...
            tempNeighbors.add(receivingNode);
            WGrid.updateWGRoutingTable(actor, receivingNode);
        }
        ArrayList<WGRoutingTableEntry> rtForLocator = WGrid.setCoordinates(actor, numberOfRealities
```

```
             , coordsPerNode ,
            coordsPerBeacon , coordsPerNeighbor ,
                 checkDependencies);
        if (rtForLocator.size() > 0)
        createdWGCoordinate = true;

        // tries to balance the number of coordinates
             among the nodes
        if (SLOB != NSParameters.WLB_OFF){
            if (WGrid.checkLoad(actor, avgThreshold,
                 sqmThreshold))
                    createdWGCoordinate = true;
        }
        for (WGReality nodeReality : actor.getWGRealities
             ())
            nodeReality.checkMaxLengthVC();
        actor.checkWGNodeMaxLengthVC();
        return createdWGCoordinate;
    }
    //load balancing (SLOB)
    private static boolean checkLoad(WGActor actor,
         double avgThreshold, double sqmThreshold){
        boolean addedCoord = false;
        for (WGReality nodeReality : actor.getWGRealities
             ()) {
            int nodeRealityId = nodeReality.getRootId();
            //calculate the average number of coordinates
                 managed by neighbors
            ArrayList<Double> values = new ArrayList<
                 Double>();
            ArrayList<Object> avgResult = calculateAvg(
                 actor, nodeRealityId, values);
            double myLoad = ((Double)avgResult.get(0)).
                 doubleValue();
            if ((NodeLoad)avgResult.get(1) == null)
                continue;
            double avg = ((Double)avgResult.get(2)).
                 doubleValue();
            double sqm = getNumCoordsSqm(values, avg);
            if (myLoad < avg) {
                if ((avg - myLoad > avgThreshold) || (avg -
```

```
          myLoad < avgThreshold && sqm >
        sqmThreshold)) {
        WGActor givingNode = ((NodeLoad) avgResult.get(1)).node;
        WGReality givingNodeReality = givingNode.getWGReality(nodeReality);
        ArrayList<WGCoordinate> candidateCoords = givingNodeReality.getGivableCoordinates();
        if (candidateCoords.size() == 0)
            candidateCoords = givingNodeReality.getNotSplitCoordinates();
        WGCoordinate givingCoord = WGrid.chooseCoordinate(actor, nodeRealityId, candidateCoords);
        if (givingCoord == null)
            continue;
        if (givingCoord.isGivable()) {
            //remove rte from reality and add the new rte to the other node
            WGRoutingTableEntry newRte = new WGRoutingTableEntry(actor, nodeRealityId, givingCoord);
            givingNodeReality.addToRoutingTable(newRte);
            nodeReality.removeFromRoutingTable(givingNode.getMobileNodeId(), givingCoord.getVC());
            //add coordinate to the node and remove it from the other node
            if (!nodeReality.moveCoordinate(actor, givingCoord))
                throw new RuntimeException("Exception␣at␣load␣balancing,␣cannot␣move␣coordinate␣" + givingCoord.getVC() + "from␣node␣" + givingCoord.getOwner() + "␣to␣node␣" + actor.getMobileNodeId());
          }
        else {
```

```
                        WGrid.fullSplit(givingNode, givingCoord, nodeRealityId, actor);
                        return true;
                    }
                }
            }
        }
        return addedCoord;
    }

    private static void updateWGRoutingTable(WGActor actor, WGActor neighbor){
        for (WGReality neighborReality : neighbor.getWGRealities()){//get its realities
            int nodeRealityId = neighborReality.getRootId();
            WGReality nodeReality = actor.getWGReality(neighborReality);
            if(nodeReality == null){//if the reality in unknown, get it
                WGReality newReality = new WGReality(nodeRealityId);
                //for each coordinate of the neighbor, put it in the routing table
                for (WGCoordinate c : neighborReality.getAllCoordinates()){
                    WGRoutingTableEntry newRte = new WGRoutingTableEntry(neighbor, nodeRealityId, c);
                    newReality.addToRoutingTable(newRte);
                }
                actor.addToWGRealities(newReality);
            }
            else{//if the reality is known, update it
                WGRoutingTableEntry newRte;
                for (WGCoordinate c : neighborReality.getAllCoordinates()){
                    WGReality myreality = actor.getWGReality(neighborReality);
                    boolean found = false;
```

```
                    for (WGRoutingTableEntry rte : myreality.getRoutingTable()){
                        if (rte.getVC() == c.getVC()){
                            rte.update();
                            found = true;
                            break;
                        }
                    }
                    if (!found){
                        newRte = new WGRoutingTableEntry(neighbor, nodeRealityId, c);
                        nodeReality.addToRoutingTable(newRte);
                    }
                }
            }
        }
    }//updateRoutingTable

    //When a node has no neighbors with w-grid coordinates it sets itself as root
    public static void setMyselfAsRoot(WGActor act){
        WGReality r = new WGReality(act.getMobileNodeId());
        WGCoordinate newC = new WGCoordinate(act.getMobileNodeId(), "*", act, act);
        r.addCoordinate(newC);
        act.addToWGRealities(r);
        act.mobileNodeIsActive(true);
    }

    //coordinate creation
    protected static ArrayList<WGRoutingTableEntry> setCoordinates(WGActor actor, int numberOfRealities, int coordsPerNode,
            int coordsPerBeacon, int coordsPerNeighbor, boolean checkDependencies){
        //if a number of root is not specified any node that has no neighbor with coordinate must elect
        //itself as root, the variable isRoot is needed
```

```
        to check that
        boolean isRoot = true;
        ArrayList<WGRoutingTableEntry> rtForLocator = new ArrayList<WGRoutingTableEntry>();

        //after having updated the r.t. the node must scan each known reality to check if it needs any coordinate
        for (WGReality nodeReality : actor.getWGRealities()){
            //if it enters actor loop it means that the node does not have to become root of a new reality
            ArrayList<WGRoutingTableEntry> coordsPerReality = new ArrayList<WGRoutingTableEntry>();
            int takenCoordPerReality = 0;
            isRoot = false;
            int coordinatesPerBeaconing = 0;
            int nodeRealityId = nodeReality.getRootId();
            //Case of k-coordinates, check if K has been reached
            if (coordsPerNode != WGrid.ONE_PER_NEIGHBOR && nodeReality.getNotSplitCoordinatesSize() < coordsPerNode) {
                while (nodeReality.getNotSplitCoordinatesSize() < coordsPerNode) {
                    //if the maximum number of coordinates per beaconing has been fixed and is reached then break the loop
                    if (coordsPerBeacon != 0 && coordinatesPerBeaconing == coordsPerBeacon)
                        break;
                    /*ArrayList that will contain the choosable coordinates, givable ones will be preferred, alway according to choosing policies (see chooseCoordinate method)*/
                    ArrayList<WGCoordinate>
```

```
    candidateCoordinates = new ArrayList<WGCoordinate>();
//Populate arraylist with neighbors' coordinates
for(WGActor neighbor : actor.getNeighbors()){
    //if a maximum number of coordinates gettable from each node has been set
    //then skip neighbors that have reached the limit
    if (coordsPerNeighbor != 0 && nodeReality.coordinatesPerNeighbor(neighbor.getMobileNodeId()) >= coordsPerNeighbor)
        continue;
    //otherwise gather the available coords (not spli ones)
    WGReality neighReality = neighbor.getWGReality(nodeRealityId);
    if (neighReality != null) {
        candidateCoordinates.addAll(neighReality.getAllNotSplitCoordinates());
    }
}
//Remove coordinates children of mine
WGrid.checkChildConstraint(candidateCoordinates, nodeReality);
//Check for coordinates dependency, if required by user
if (checkDependencies) {
    WGrid.checkForDependencies(actor, candidateCoordinates,nodeReality);
}
if (candidateCoordinates.size() > 0) {
    WGCoordinate candidateCoord = WGrid.chooseCoordinate(actor, nodeRealityId,
```

```
                candidateCoordinates);
            coordsPerReality.add(WGrid.addCoordinate(actor, candidateCoord, nodeRealityId));
            coordinatesPerBeaconing++;
        }
        else
            break;
    }
}
else{
    if (coordsPerNode == WGrid.ONE_PER_NEIGHBOR){
        ArrayList<WGActor> missingNodes = nodeReality.checkIfHasOnePerNeighbor(actor);
        if(missingNodes.size() > 0) {
            for(WGActor neighbor : missingNodes){
                //ConsolePanel.getInstance().printToConsole("Scorro i vicini da cui devo prendere...");
                ArrayList<WGCoordinate> candidateCoordinates = new ArrayList<WGCoordinate>();
                WGReality neighReality = neighbor.getWGReality(nodeRealityId);
                if (neighReality != null)
                    candidateCoordinates.addAll(neighReality.getAllNotSplitCoordinates());
                //Remove coordinates children of mine
                checkChildConstraint(candidateCoordinates, nodeReality);
                //Check for coordinates dependency, if required by user
                if (checkDependencies)
```

```
                                WGrid.checkForDependencies(actor, candidateCoordinates,nodeReality);
                            if (candidateCoordinates.size() > 0) {
                                WGCoordinate candidateCoord = WGrid.chooseCoordinate(actor, nodeRealityId, candidateCoordinates);
                                coordsPerReality.add(WGrid.addCoordinate(actor, candidateCoord, nodeRealityId));
                            }
                            else
                                continue;
                        }
                    }
                }
            }
            for (WGRoutingTableEntry rte : coordsPerReality){
                rtForLocator.add(rte);
                takenCoordPerReality++;
            }
        }
        if(isRoot)
            if(numberOfRealities == 0 && actor.getWGReality(actor.getMobileNodeId()) == null)
                WGrid.setMyselfAsRoot(actor);
        return rtForLocator;
    }

    private static void checkChildConstraint(ArrayList<WGCoordinate> candidateCoordinates, WGReality nodeReality) {
        int i = 0;
        while (i < candidateCoordinates.size()) {
            for (WGCoordinate c : nodeReality.getAllCoordinates()) {
```

```
                if (candidateCoordinates.get(i).getVC().startsWith(c.getVC())) {
                    candidateCoordinates.remove(i);
                    i--;
                    break;
                }
            }
            i++;
        }
    }

    private static WGRoutingTableEntry addCoordinate(WGActor actor, WGCoordinate candidateCoord, int nodeRealityId){
        WGReality nodeReality = actor.getWGReality(nodeRealityId);
        if (candidateCoord.isGivable()){
            //remove rte from reality and add the new rte to the other node
            WGRoutingTableEntry newRte = new WGRoutingTableEntry(actor,nodeRealityId, candidateCoord);
            candidateCoord.getOwner().getWGReality(nodeRealityId).addToRoutingTable(newRte);
            nodeReality.removeFromRoutingTable(candidateCoord.getOwner().getMobileNodeId(), candidateCoord.getVC());
            //add coordinate to the node and remove it from the other node
            if (!nodeReality.moveCoordinate(actor, candidateCoord))
                throw new RuntimeException("Exception at split, cannot move coordinate " + candidateCoord + " from node " + candidateCoord.getOwner() + " to node " + actor.getMobileNodeId());
                return newRte;
        }
        else //split candidateCoord
            return WGrid.fullSplit(candidateCoord.getOwner(), candidateCoord, nodeRealityId, actor);
```

```
    }

    //coordinate selection
    private static void checkForDependencies(WGActor actor, ArrayList<WGCoordinate> candidateCoords, WGReality nodeReality) {
        ArrayList<WGCoordinate> neighborsCoordinates = new ArrayList<WGCoordinate>();
        for(WGActor neighbor : actor.getNeighbors()){
            if (neighbor.getMobileNodeId() == nodeReality.getRootId())
                continue;
            WGReality neighReality = neighbor.getWGReality(nodeReality);
            if (neighReality != null) {
                for (WGCoordinate c : neighReality.getAllCoordinates()) {
                    neighborsCoordinates.add(c);
                }
            }
        }
        for (WGCoordinate c : actor.getWGReality(nodeReality).getAllCoordinates()) {
            neighborsCoordinates.add(c);
        }
        int i = 0;
        while (i < candidateCoords.size()) {
            WGCoordinate cc = candidateCoords.get(i);
            i++;
            for (WGCoordinate c : neighborsCoordinates){
                if (cc.isRelatedWith(c.getVC()) && c.getOwner() != cc.getOwner()) {
                    candidateCoords.remove(--i);
                    break;
                }
            }
        }
    }

    private static WGCoordinate chooseCoordinate(WGActor actor, int nodeRealityId, ArrayList<
```

```
     WGCoordinate> candidateCoords){
    WGReality nodeReality = actor.getWGReality(
        nodeRealityId);
    ArrayList<CoordinateTableEntry>
        notGivableCoordTable= new ArrayList<
        CoordinateTableEntry>();
    ArrayList<CoordinateTableEntry> givableCoordTable
        = new ArrayList<CoordinateTableEntry>();
    //scan routing table looking for the node that
        would return the shorter CV
    //result will store the candidates vc (the
        shorter ones)
    int maxLength = 0;
    int minLength = 255;
    int maxLoad = 0;
    int maxHeterogenity = 0;
    int minHeterogenity = 0x7fffffff;
    ArrayList<Integer> heterogeneities = new
        ArrayList<Integer>();
    ArrayList<Integer> lengths = new ArrayList<
        Integer>();
    ArrayList<Integer> loads = new ArrayList<Integer
        >();
    //first try to get a givable coord
    for (WGCoordinate coord : candidateCoords){
        //to avoid exception because length of
            freespace is greatest than
            binaryNodeIdLength
        if (coord.getVC().length()-1 >= NSParameters.
            getInstance().getBinaryIdLength())
            continue;
            CoordinateTableEntry ce = new
                CoordinateTableEntry(coord);
        if (coord.isGivable())
            givableCoordTable.add(ce);
        else
            notGivableCoordTable.add(ce);
        int length = coord.getVC().length();
        if (length > maxLength)
            maxLength = length;
        if (length < minLength)
```

```java
                minLength = length;
            lengths.add(length);
            int heterogeneityValue = 0;
            if(nodeReality.getCoordinates().size()==0){//if the node doesn't already have any coord, calculate distance among available coords
                for (WGCoordinate c1 : candidateCoords)
                    heterogeneityValue += 255 - WGTools.maxCommonPrefLength(coord.getVC(),c1.getVC());
            }
            else{//else if the node already manage some coords, evaluate distance between each of them and available ones
                for(WGCoordinate c1 : nodeReality.getNotSplitCoordinates())
                    heterogeneityValue += 255 - WGTools.maxCommonPrefLength(coord.getVC(),c1.getVC());
            }
            heterogeneities.add(heterogeneityValue);
            if (heterogeneityValue > maxHeterogenity)
                maxHeterogenity = heterogeneityValue;
            if (heterogeneityValue < minHeterogenity)
                minHeterogenity = heterogeneityValue;
            }
        //calculates length value(scaled into range [0,1])
        int lengthRange = maxLength - minLength;
        if (lengthRange != 0 && NSParameters.getInstance().getLengthFactor() != 0) {
            int i = 0;
            for (CoordinateTableEntry ce : givableCoordTable){
                double lengthValue = ((double)(maxLength - lengths.get(i))/(double)lengthRange) * NSParameters.getInstance().getLengthFactor();
                ce.setValue(lengthValue);
                i++;
```

```
            }
            for (CoordinateTableEntry ce : notGivableCoordTable){
                double lengthValue = ((double)(maxLength - lengths.get(i))/(double)lengthRange) * NSParameters.getInstance().getLengthFactor();
                ce.setValue(lengthValue);
                i++;
            }
        }
        //calculates heterogeneity value
        int heterogeneityRange = maxHeterogenity - minHeterogenity;
        if (heterogeneityRange != 0 && NSParameters.getInstance().getHeterogeneityFactor() != 0) {
            int i = 0;
            for (CoordinateTableEntry ce : givableCoordTable){
                double heterogeneityValue =  ((double)(heterogeneities.get(i) - minHeterogenity) / (double) heterogeneityRange) * NSParameters.getInstance().getHeterogeneityFactor();
                ce.setValue(heterogeneityValue);
                i++;
            }
            for (CoordinateTableEntry ce : notGivableCoordTable){
                double heterogeneityValue =  ((double)(heterogeneities.get(i) - minHeterogenity) / (double) heterogeneityRange) * NSParameters.getInstance().getHeterogeneityFactor();
                ce.setValue(heterogeneityValue);
                i++;
            }
        }
        //calculates coordinate load value
        if (maxLoad != 0 && NSParameters.getInstance().
```

```
            getLoadFactor() != 0) {
            for (CoordinateTableEntry ce : notGivableCoordTable){
                int i = 0;
                double loadValue = (loads.get(i) / maxLoad) * NSParameters.getInstance().getLoadFactor();
                ce.setValue(loadValue);
                i++;
            }
            for (CoordinateTableEntry ce : givableCoordTable){
                int i = 0;
                double loadValue = (loads.get(i) / maxLoad) * NSParameters.getInstance().getLoadFactor();
                ce.setValue(loadValue);
                i++;
            }
        }
        //extract from coordTable the most valued coord according to the choosed policy
        if (givableCoordTable.size() > 0)
            return CoordinateTableEntry.extract(givableCoordTable).getCoordinate();
        else
            if (notGivableCoordTable.size() > 0)
                return CoordinateTableEntry.extract(notGivableCoordTable).getCoordinate();
            else
                throw new RuntimeException("ArrayList is empty, cannot choose any coordinate!");
    }
    //(end)coordinate selection

    //coordinate split
    private static WGRoutingTableEntry fullSplit(WGActor actor, WGCoordinate c, int nodeRealityId, WGActor askingNode){
        if (c.hasSplit())//bug checking
            throw new RuntimeException("Coordinate " + c.
```

```
            getVC() +"␣has␣already␣been␣split");
        WGReality nodeReality = actor.getWGReality(
            nodeRealityId);
        WGRoutingTableEntry result = null;
        //new coordinates(buddies)
        WGCoordinate newC1 = new WGCoordinate(
            nodeRealityId, c.getVC()+"0",c.getOwner(),c.
            getOwner());
        WGCoordinate newC2 = new WGCoordinate(
            nodeRealityId, c.getVC()+"1",c.getOwner(),c.
            getOwner());
        //distribution of locating nodes among the
            buddies
        Iterator<String> it = c.getManagingData().
            iterator();
        while (it.hasNext()) {
            String s = it.next();
                if(WGTools.maxCommonPref(s,newC1.getVC()).
                    length() == newC1.getVC().length()){
                    newC1.addToManagingData(s);
                    it.remove();
                }
            else {
                if(WGTools.maxCommonPref(s,newC2.getVC()).
                    length() == newC2.getVC().length()) {
                    newC2.addToManagingData(s);
                    it.remove();
                }
            }
        }
        //set c as split coordinate, //c becomes not
            givable
        if (!nodeReality.setSplit(c,askingNode))
            throw new RuntimeException("Exception␣at␣split
                ,␣cannot␣split␣coordinate␣"+c.getVC());
        if (c.isGivable()){
            /*if c is givable the split is due to an
                overflow
            add buddies to node Reality(to givable coords
                list)*/
            nodeReality.addGivableCoordinate(newC1);
```

```
            nodeReality.addGivableCoordinate(newC2);
            //check if overflow situation is recovered on
                both buddies
            if (newC1.getManagingDataSize() > NSParameters
                .getInstance().getBucketSize()){
                WGrid.fullSplit(actor, newC1,nodeRealityId,
                    null);
            }
            if (newC2.getManagingDataSize() > NSParameters
                .getInstance().getBucketSize()){
                WGrid.fullSplit(actor, newC2,nodeRealityId,
                    null);
            }
        }
        else {//else check if it is an overflow or a new
            Coordinate creation
            int bit = WGTools.returnBit();
            if (askingNode == null) {//case of overflow //
                checked: OK
                if (bit == 0) {
                    nodeReality.addCoordinate(newC1);
                    nodeReality.addGivableCoordinate(newC2);
                }
                else {
                    nodeReality.addGivableCoordinate(newC1);
                    nodeReality.addCoordinate(newC2);
                }
                //check if overflow situation is recovered
                    on both buddies
                if (newC1.getManagingDataSize() >
                    NSParameters.getInstance().
                    getBucketSize())
                    WGrid.fullSplit(actor, newC1,
                        nodeRealityId,null);
                if (newC2.getManagingDataSize() >
                    NSParameters.getInstance().
                    getBucketSize())
                    WGrid.fullSplit(actor, newC2,
                        nodeRealityId,null);
            }
            else{//new coordinate
```

```
                WGCoordinate givingCoord, stillCoord;
                WGReality askingNodeReality = askingNode.
                     getWGReality(nodeRealityId);
                if (bit == 0) {
                    givingCoord = newC1;
                    stillCoord = newC2;
                }
                else {
                    stillCoord = newC1;
                    givingCoord = newC2;
                }
                givingCoord.setOwner(askingNode);
                //add coordinate to the respective node
                nodeReality.addCoordinate(stillCoord);
                askingNodeReality.addCoordinate(givingCoord
                     );
                //update routing table
                result = new WGRoutingTableEntry(askingNode
                     , nodeRealityId, givingCoord);
                nodeReality.addToRoutingTable(result);
                askingNodeReality.addToRoutingTable(new
                     WGRoutingTableEntry(actor,
                     nodeRealityId, stillCoord));
            }
        }
        nodeReality.checkMaxLengthVC();
        actor.checkWGNodeMaxLengthVC();
        return result;
    }

    private static ArrayList<Object> calculateAvg(
         WGActor actor, int nodeRealityId, ArrayList<
         Double> values){
        double avg = 0d;
        double myLoad = 0d;
        double load = 0d;
        double value = 0d;
        //Result arrayList contains:
        //1) My load
        //2) Most loaded node
        //3) Avg load
```

```
ArrayList<Object> result = new ArrayList<Object>(3);
WGReality nodeReality = actor.getWGReality(nodeRealityId);
if (NSParameters.getInstance().useWGLoadBalancing() == NSParameters.WLB_NE_COORDINATES_NUMBER)
    value = (double)nodeReality.getNotEmptyCoordinatesSize();
if (NSParameters.getInstance().useWGLoadBalancing() == NSParameters.WLB_COORDINATES_NUMBER)
    value = (double)nodeReality.getAllNotSplitCoordinatesSize();
if (NSParameters.getInstance().useWGLoadBalancing() == NSParameters.WLB_DATA_SPACE)
    value = nodeReality.getSpacePortion();
//actor so far is myLoad
result.add(value);
load += value;
myLoad = value;
values.add(value);
//check each neigbors, calculate avgLoad and sort nodes that have a higher load than mine
ArrayList<NodeLoad> mostLoaded = new ArrayList<NodeLoad>();
WGReality neighborReality = null;
for (WGActor n : nodeReality.getDistinctNeighbors()) {
    if (!n.isMobileNodeActive())
        continue;
    neighborReality = n.getWGReality(nodeRealityId);
    if (neighborReality == null)
        continue;
    if (NSParameters.getInstance().useWGLoadBalancing() == NSParameters.WLB_NE_COORDINATES_NUMBER)
        value = (double)neighborReality.getNotEmptyCoordinatesSize();
    if (NSParameters.getInstance().useWGLoadBalancing() == NSParameters.WLB_COORDINATES_NUMBER)
```

```
                value = (double)neighborReality.getAllNotSplitCoordinatesSize();
            if (NSParameters.getInstance().useWGLoadBalancing() == NSParameters.WLB_DATA_SPACE)
                value = neighborReality.getSpacePortion();
            if (value > myLoad) {
                int i = 0;
                while (i < mostLoaded.size() && mostLoaded.get(i).load > value) {
                    i++;
                }
                mostLoaded.add(i,new NodeLoad(n,load));
            }
        }
        WGrid.checkChildConstraintOnBal(mostLoaded, nodeReality);
        WGrid.extractMostLoadedNodesl(mostLoaded);
        if (mostLoaded.size() == 0) {
            result.add(null);
        }else {
            result.add(mostLoaded.get(WGRandom.getInstance().nextInt(mostLoaded.size())));
        }
        WGCoordinate coord = null;
        WGActor n = null;
        if ((NodeLoad)result.get(1) != null){
            for (WGCoordinate c : nodeReality.getAllCoordinates()) {
                if (c.getNodeFatherId() == ((NodeLoad)result.get(1)).node.getMobileNodeId()) {
                    coord = c;
                }
            }
            if (coord != null) {
                String vc = coord.getVC();
                int i = 0;
                WGReality nr = nodeReality;
                while (vc.length() > 1 && i < 3) {
                    while (coord.getNodeFatherId() == actor.
```

```
                 getMobileNodeId() && vc.length() >1)
                  {
                vc = vc.substring(0,vc.length()-1);
            }
            if (vc.length() ==1)
                break;
            coord = nr.getCoordinateFromVC(vc);
            n = coord.getNodeFather();
            nr = n.getWGReality(nodeRealityId);
            if (nr != null) {
                vc = vc.substring(0,vc.length()-1);
                coord = nr.getCoordinateFromVC(vc);
            }else
                break;
            i++;
        }
        if (i == 3) {
            boolean neighbor = false;
            for (WGActor n1 : nodeReality.
                 getDistinctNeighbors()) {
                if (n1.getMobileNodeId() == n.
                     getMobileNodeId()) {
                    neighbor = true;
                    break;
                }
            }
            if (!neighbor) {
                if (NSParameters.getInstance().
                     useWGLoadBalancing() ==
                     NSParameters.
                     WLB_NE_COORDINATES_NUMBER)
                    value = (double)nr.
                         getNotEmptyCoordinatesSize();
                if (NSParameters.getInstance().
                     useWGLoadBalancing() ==
                     NSParameters.
                     WLB_COORDINATES_NUMBER)
                    value = (double)nr.
                         getAllNotSplitCoordinatesSize
                         ();
                if (NSParameters.getInstance().
```

```
                      useWGLoadBalancing() ==
                      NSParameters.WLB_DATA_SPACE)
                     value = nr.getSpacePortion();
                  if (value > 0d) {
                     value += value;
                     values.add(value);
                  }
               }
            }
         }
      }
      avg = ((double)load/(double)values.size());
      result.add(avg);
      return result;
   }

   private static double getNumCoordsSqm(ArrayList<
       Double> values, double avg) {
      double sqm = 0d;
      for (Double value : values)
         sqm += Math.pow(value - avg, 2);
      sqm = Math.sqrt(sqm/(double)values.size());
      return sqm ;
   }

   /*mostloaded holds all the nodes which have higher
       load than current node,
   actor method truncate the list to the most loaded
       one(s)*/
   private static void extractMostLoadedNodesl(
       ArrayList<NodeLoad > mostLoaded) {
      double maxLoad = 0d;
      if (mostLoaded.size() == 0)
         return;
      maxLoad = mostLoaded.get(0).load;
      int i = 1;
      while (i < mostLoaded.size() && mostLoaded.get(i)
          .load == maxLoad)
         i++;
      while (i < mostLoaded.size())
         mostLoaded.remove(i);
```

```
}

//Used a different name for this method since java
    does not distinguish between arrayLists of
    different types
private static void checkChildConstraintOnBal(
    ArrayList<NodeLoad > mostLoaded, WGReality
    nodeReality) {
  int index  = 0;
  while (index < mostLoaded.size()) {
    WGReality neighReality = mostLoaded.get(index)
        .node.getWGReality(nodeReality);
    if (neighReality == null)
      continue;
    ArrayList<WGCoordinate> candidateCoordinates =
          neighReality.getAllNotSplitCoordinates();
    int i = 0;
    while (i < candidateCoordinates.size()) {
      for (WGCoordinate c : nodeReality.
          getAllCoordinates()) {
        if (candidateCoordinates.get(i).getVC().
            startsWith(c.getVC())) {
          candidateCoordinates.remove(i);
          i--;
          break;
        }
      }
      i++;
    }
    //if none of the coordinates satisfies
        constraint take node out from most loaded
        array
    if (candidateCoordinates.size() == 0) {
      mostLoaded.remove(index);
    }else
    index++;
  }
}

public static void sendQuery(WGActor actor, String
    recipientVC){
```

```
        WGPacket d = new WGPacket(WGPacket.QUERY, actor, actor.getWGRealities().get(0).getRootId(), recipientVC);
        receive(actor, d);
        return;
    }

    public static void receive(WGActor actor, WGPacket d){
        actor.incWGReceivedPackets();
        ArrayList next = WGrid.route(actor, d);
        WGActor nextNode = (WGActor)next.get(0);
        int nextReality = (Integer)next.get(1);
        String nextVC = (String)next.get(2);
        int myReality = (Integer)next.get(3);
        String myVC = (String)next.get(4);
        if (d.getLastCrossedNodeId() != actor.getMobileNodeId())
            d.addToHistory(new WGPacketHistoryEntry(actor, myReality, myVC));
        if (d.getPacketType() == WGPacket.RECOVERY_PACKET){
            if (nextNode.getMobileNodeId() == actor.getMobileNodeId()){
                for (WGCoordinate c : actor.getWGReality(d.getRoutingRealityId()).getSplitCoordinates()){
                    if (c.getVC().equals(d.getDestinationVC().substring(0, d.getDestinationVC().length()-1))){
                        d.setStatus(WGPacket.DELIVERED);
                    d.setRecipientId(actor.getMobileNodeId());
                    d.handle();
                    return;
                    }
                }
                d.setStatus(WGPacket.DROPPED);
                d.handle();
                return;
            }
```

```
            return WGrid.receive(nextNode, d);
        }
        if (d.getPacketType() == WGPacket.DATA_INSERTION) {
            if (nextNode.getMobileNodeId() == actor.getMobileNodeId()){
                WGCoordinate targetCoord = actor.getWGReality(nextReality).getCoordinateFromVC(nextVC);
                if (targetCoord == null || targetCoord.hasSplit() || !d.getDestinationVC().startsWith(targetCoord.getVC())){
                    d.setStatus(WGPacket.DROPPED);
                    d.handle();
                    return;
                }
            else{
                d.setStatus(WGPacket.DELIVERED);
                d.setRecipientId(actor.getMobileNodeId());
                targetCoord.addToManagingData(d.getDestinationVC());
                if (targetCoord.getManagingDataSize() > NSParameters.getInstance().getBucketSize() && NSParameters.getInstance().useWGLoadBalancing() != NSParameters.WLB_OFF){
                    WGrid.fullSplit(actor, targetCoord, nextReality ,null);
                }
                d.handle();
                return;
            }
            return WGrid.receive(nextNode, d);
        }
        if (d.getPacketType() == WGPacket.QUERY){
            String coord = "";
            //Real distance (RD)
            if (d.getHistorySize() > 1){
                int as = 0;
                float totNearing = avgNearing * numNearing;
                for (int i = 0; i < d.getHistorySize() -1;
```

```
                    i++){
                   WGPacketHistoryEntry p1 = d.
                       getHistoryEntry(i);
                   WGPacketHistoryEntry p2 = d.
                       getHistoryEntry(i+1);
                   if (p1.getRealityId() == p2.getRealityId
                       ()){
                       as = WGTools.WGDistance(p1.
                           getCoordinate(),p2.getCoordinate
                           ());
                       totNearing += as;
                       numNearing++;
                   }
               }
               avgNearing = totNearing / numNearing;
           }
           //local learning (LL)
           int distance = (Integer)next.get(5);
           boolean local = false;
           WGActor tempNext = null;
           int minDistance = 0x7fffffff;
           int tempDistance = 0x7fffffff;
           for (WGActor n : actor.getNeighbors()){
               if (!actor.isMobileNodeActive())
                   continue;
               if (n.getMobileNodeId() == nextNode.
                   getMobileNodeId())
                   continue;
               WGReality nr = n.getWGReality(d.
                   getRoutingRealityId());
               for (WGRoutingTableEntry rte : nr.
                   getRoutingTable()) {
                   if (!rte.getReferredNode().
                       isMobileNodeActive())
                   continue;
                   if (rte.getReferredNode().hasNeighbor(
                       nextNode.getMobileNodeId()))
                       continue;
                   tempDistance = WGTools.WGDistance(rte.
                       getVC(), d.getDestinationVC());
                   if (tempDistance < distance-1 &&
```

```
                        tempDistance < minDistance) {
                        local = true;
                        minDistance = tempDistance;
                        tempNext = n;
                        coord = rte.getVC();
                    }
                    if (rte.getNBit() > 0){
                        int i = rte.getNBit();
                        while (i < rte.getVC().length()){
                            //String s = rte.getVC().substring(0, rte.getNBit());
                            tempDistance = WGTools.WGDistance(rte.getVC().substring(0, rte.getNBit()),d.getDestinationVC()) +1;
                            if (tempDistance < distance-1 && tempDistance < minDistance) {
                                local = true;
                                minDistance = tempDistance;
                                tempNext = n;
                                coord = rte.getVC();
                            }
                            i++;
                        }
                    }
                }
            }
            if (local){//if a local has been found store it
                nextNode = tempNext;
                nextVC = coord;
            }
            if (nextNode.getMobileNodeId() == actor.getMobileNodeId()){
                WGCoordinate targetCoord = actor.getWGReality(nextReality).getCoordinateFromVC(nextVC);
                if (targetCoord == null || nextReality ==-1 || !d.getDestinationVC().startsWith(nextVC) || d.getDestinationVC().startsWith(nextVC)&&targetCoord.
```

```
                    hasSplit()) {
                    d.setStatus(WGPacket.DROPPED);
                    d.handle();
                    return;
                }
                else{
                    if (targetCoord.hasWGData(d.getDestinationVC())){
                        d.setStatus(WGPacket.DELIVERED);
                        d.setRecipientId(actor.getMobileNodeId());
                    }
                    else
                        d.setStatus(WGPacket.DROPPED);
                    d.handle();
                    return;
                }
            }
            return WGrid.receive(nextNode, d);
        }
    }

    //used during a dataPacket routing to find the next node if no learning can be used
    private static ArrayList route(WGActor actor, WGPacket d){
        //result contains:
        //(0) best successor node
        //(1) reality of closest VC to destination
        //(2) coord of closest VC to destination
        //(3) reality of my closest VC to destination
        //(4) coord of my closest VC to destination
        //(5) logical distance
        ArrayList<Object> result = new ArrayList<Object>(5);
        WGRoutingTableEntry next = null;//new WGRoutingTableEntry(actor,nodeRealityId,vc);
        int myDistance = 0xfffffff;
        int myRealityId = -1;
        int minDistance = myDistance;
        String myVC = "";
```

```
if (d.getRecipientVCs() != null && d.getRecipientVCs().size() >0){
    //coordinates are known
    for (WGReality r : actor.getWGRealities()) {
        for(WGCoordinate c : r.getAllCoordinates()) {
            if (d.getRav(r.getRootId()) != null){
                if (d.getRavD(r.getRootId()) <= WGTools.WGDistance(c.getVC(), d.getDestinationVC()))
                    continue;
            }
            //find my vc with max common prefix with destination
            for (WGRoutingTableEntry rrte : d.getRecipientVCs()) {
                if (rrte.getRealityId() != r.getRootId())
                    continue;
                int tempDistance = WGTools.WGDistance(c.getVC(), rrte.getVC());
                if (tempDistance < myDistance) {
                    myDistance = tempDistance;
                    myRealityId = r.getRootId();
                    myVC = c.getVC();
                }
            }
        }
    }
    minDistance = myDistance;
    for (WGReality r : actor.getWGRealities()){
        for (WGRoutingTableEntry rte : r.getRoutingTable()) {
            if (d.getRav(r.getRootId()) != null){
                if (d.getRavD(r.getRootId()) <= WGTools.WGDistance(rte.getVC(), d.getDestinationVC()))
                    continue;
            }
            if (!rte.getReferredNode().isMobileNodeActive())
```

```
                        continue;
                    for (WGRoutingTableEntry rrte : d.getRecipientVCs()){
                        if (rrte.getRealityId() != r.getRootId())
                            continue;
                        int tempDistance = WGTools.WGDistance(rte.getVC(), rrte.getVC());
                        if (tempDistance < minDistance) {
                            minDistance = tempDistance;
                            next = rte;
                        }
                    }
                }
            }
        }
        else { //coordinates are unknown, must travel toward destinationVC
            int routingRealityId = d.getRoutingRealityId();
            if (routingRealityId != -1) {
                WGReality nodeReality = actor.getWGReality(routingRealityId);
                for(WGCoordinate c : nodeReality.getAllCoordinates()){
                    //find my vc with max common prefix with destination
                    if (d.getPrefixToAvoid().startsWith("*") && c.getVC().startsWith(d.getPrefixToAvoid()))
                        continue;
                    int tempDistance = WGTools.WGDistance(c.getVC(), d.getDestinationVC());
                    if (tempDistance < myDistance) {
                        myDistance = tempDistance;
                        myVC = c.getVC();
                        myRealityId = routingRealityId;
                    }
                }
                minDistance = myDistance;
                for (WGRoutingTableEntry rte : nodeReality.
```

```
                    getRoutingTable()) {
                    /*It is neccessary to work, infact only the node that started
                    the beaconing has certainly adjusted its r.t.. Its neighbors might have not ...*/
                    if (!rte.getReferredNode().isMobileNodeActive())
                        continue;
                    if (d.getPrefixToAvoid().startsWith(”*”) && rte.getVC().startsWith(d.getPrefixToAvoid()))
                        continue;
                    if (d.getNodeToAvoid() == rte.getReferredNodeId())
                        continue;

                    int tempDistance = WGTools.WGDistance(rte.getVC(),d.getDestinationVC());
                    if (tempDistance < minDistance) {
                        minDistance = tempDistance;
                        next = rte;
                    }
                    if (rte.getNBit() > 0){
                        int i = rte.getNBit();
                        while (i < rte.getVC().length()){
                            //String s = rte.getVC().substring(0, rte.getNBit());
                            tempDistance = WGTools.WGDistance(rte.getVC().substring(0, rte.getNBit()),d.getDestinationVC()) +1;
                            if (tempDistance < minDistance) {
                                minDistance = tempDistance;
                                next = rte;
                            }
                            i++;
                        }
                    }
                }
            }
```

```
else {
    for (WGReality r : actor.getWGRealities()) {
        for(WGCoordinate c : r.getAllCoordinates()){
            if (d.getRav(r.getRootId()) != null){
                if (d.getRavD(r.getRootId()) <= WGTools.WGDistance(c.getVC(), d.getDestinationVC()))
                    continue;
            }
        //find my vc with max common prefix with destination
            int tempDistance = WGTools.WGDistance(c.getVC(), d.getDestinationVC());
            if (tempDistance < myDistance) {
                myDistance = tempDistance;
                myRealityId = r.getRootId();
                myVC = c.getVC();
            }
        }
    }
    minDistance = myDistance;
    for (WGReality r : actor.getWGRealities()){
        for (WGRoutingTableEntry rte : r.getRoutingTable()) {
            if (d.getRav(r.getRootId()) != null){
                if (d.getRavD(r.getRootId()) <= WGTools.WGDistance(rte.getVC(), d.getDestinationVC()))
                    continue;
            }
            if (!rte.getReferredNode().isMobileNodeActive())
            continue;
            int tempDistance = WGTools.WGDistance(rte.getVC(),d.getDestinationVC());
            if (tempDistance < minDistance) {
                minDistance = tempDistance;
```

```
                            next = rte;
                        }
                    }
                }
            }
        }
        if (minDistance < myDistance) {
            result.add(next.getReferredNode());
            result.add(next.getRealityId());
            result.add(next.getVC());
            result.add(myRealityId);
            result.add(myVC);
            result.add(minDistance);
        }
        else{
            result.add(actor);
            result.add(myRealityId);
            result.add(myVC);
            result.add(myRealityId);
            result.add(myVC);
            result.add(myDistance);
        }
            return result;
    }

    //learned routing table
    public static boolean hasNeighbor(WGActor actor, int n) {
        for (WGReality r : actor.getWGRealities()) {
            for (WGRoutingTableEntry rte : r.getRoutingTable()) {
                if (rte.getReferredNodeId() == n)
                return true;
            }
        }
        return false;
    }

}

//Inner class used to evaluate the coordinates among
```

```
    which node must choose one
class CoordinateTableEntry{
   WGCoordinate coordinate;
   double value;

   CoordinateTableEntry(WGCoordinate c){
      coordinate = c;
      value = 0d;
   }
   CoordinateTableEntry(CoordinateTableEntry cte){
      coordinate = cte.getCoordinate();
      value = cte.getValue();
   }
   public WGCoordinate getCoordinate() {
      return coordinate;
   }
   public double getValue() {
      return value;
   }
   public  void setValue(double v) {
      value += v;
   }
   static CoordinateTableEntry extract(ArrayList<
       CoordinateTableEntry> coordTable) {
      for (int k = 1; k < coordTable.size(); k++)
         for (int i = 0; i < coordTable.size()-k; i++){
            CoordinateTableEntry cte1 = coordTable.get(
                i);
            CoordinateTableEntry cte2 = coordTable.get(
                i+1);
               if (cte1.getValue() > cte2.getValue()) {
                  CoordinateTableEntry tempCte = new
                      CoordinateTableEntry(cte1);
                  coordTable.set(i,cte2);
                  coordTable.set(i+1,tempCte);
               }
         }
      ArrayList<CoordinateTableEntry> result = new
          ArrayList<CoordinateTableEntry >();
      double maxValue = coordTable.get(coordTable.size
          ()-1).getValue();
```

```
        result.add(coordTable.get(coordTable.size()-1));
        int i = coordTable.size()-2;
        while (i >0 && coordTable.get(i).getValue() ==
            maxValue) {
            result.add(coordTable.get(i));
            i--;
        }
        return result.get(WGRandom.getInstance().nextInt(
            result.size()));
    }
}

//Inner class NodeLoad
class NodeLoad{
    NodeLoad(WGActor n, double l) {
        node = n;
        load = l;
    }
    WGActor node;
    double load;
}
```

Class WGtools

```
public class WGTools{
   /**
   * Retuns the distance between two W-grid coordinates
   * @param coord1 The first coordinate.
   * @param coord2 The second coordinate.
   * @return The distance between the two W-grid
       coordinates.
   */

   public static int WGDistance(String coord1, String
       coord2) {
      int mcpLength = WGTools.maxCommonPref(coord1,
         coord2).length();
      return (coord1.length()-mcpLength) + (coord2.
         length()-mcpLength);
   }//distance

   public static String findVC(int x, int y, int maxX,
       int maxY, int length){
      String vc = "*";
        String sx = "", sy = "";
        int lx = length / 2 + length % 2;
        int ly = length / 2 ;
        int powX = (int)Math.pow(2, lx);
        int powY = (int)Math.pow(2, ly);
        int i = (int)(((double)x / maxX) * powX);
      do{
            sx = i % 2 + sx;
            i = i / 2;
        }
        while (i != 0);
      while (sx.length()<lx)
                sx = "0" + sx;
        int j = (int)(((double)y / maxY)*powY);
      do{
            sy = j % 2 + sy;
            j = j / 2;
        }
        while (j != 0);
```

```
        while (sy.length()<ly)
            sy = "0" + sy;
        while (sx.length() >0){
            vc += sx.charAt(0);
            sx = sx.substring(1);
            if (sy.length() > 0){
                pi += sy.charAt(0);
                sy = sy.substring(1);
            }
        }
        return vc;
    }

}
```

Bibliography

[1] *Wireless LAN medium access control (MAC) and physical layer (PHY) specifications, IEEE standard 802.11, June 1999.*

[2] R. Bischoff and R. Wattenhofer. Analyzing connectivity-based multi-hop ad-hoc positioning. In *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, page 165, Washington, DC, USA, 2004. IEEE Computer Society.

[3] W. Liu C. Chiang, H. Wu and M. Gerla. Routing in clustered multihop, mobile wireless networks. In *Proc. IEEE SICON'97*, pages 197–211, April 1997.

[4] Jakob Eriksson, Michalis Faloutsos, and Srikanth Krishnamurthy. Peernet: Pushing peer-to-peer down the stack. In *IPTPS*, 2003.

[5] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[6] B. Greenstein, D. Estrin, R. Govindan, S. Ratnasamy, and S. Shenker. Difs: A distributed index for features in sensor networks. In *Proceedings of first IEEE WSNA*, pages 163–173. IEEE Computer Society, 2003.

[7] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[8] Z. J. Haas and M. R. Pearlman. The zone routing protocol: A hybrid framework for routing in ad hoc networks. *Ad Hoc Networks*, 2000.

[9] C. Hedrick. Routing information protocol. Technical Report Internet Request for Comments 1058, 1988.

[10] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw.*, 11(1):2–16, 2003.

[11] A. Iwata, C.-C. Chiang, G. Pei, M. Gerla, , and T.-W. Chen. Scalable routing strategies for ad hoc wireless networks. *IEEE Journal on Selected Areas in Communications*, 1999.

[12] D.B. Johnson, D.A. Maltz, and J. Broch. DSR: the dynamic source routing protocol for multihop wireless ad hoc networks. *Ad hoc networking*, pages 139–172, 2001.

[13] B. Karp and H.T. Kung. GPRS: greedy perimeter stateless routing for wireless networks. In *MobiCom '00: 6th annual international conference on Mobile computing and networking*, pages 243–254. ACM Press, 2000.

[14] F. Kuhn, R. Wattenhofer, Y. Zhang, and A. Zollinger. Geometric adhoc routing: of theory and practice. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 63–72. ACM Press, 2003.

[15] X. Li, Y.J. Kim, R. Govindan, and W. Hong. Multi-dimensional range queries in sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 63–75, New York, NY, USA, 2003. ACM Press.

[16] K. Menger. Zur allgemeinen kurventhoerie. *Fund. Math.*, 10:96–115, 1927.

[17] G. Monti, G. Moro, and S. Lodi. W*-Grid a robust decentralized crosslayer infrastructure for routing and multi-dimensional data management in wireless ad-hoc sensor networks. In *P2P 2007: Seventh IEEE International Conference on Peer-To-Peer Computing*, 2007.

[18] G. Monti, G. Moro, and C. Sartori. W^R-Grid: A scalable cross-layer infrastructure for routing, multi-dimensional data management and replication in wireless sensor networks. In *ISPA 2006: International Symposium on Parallel and Distributed Processing and Application*, 2006.

[19] G. Moro and G. Monti. W-Grid: a self-organizing infrastructure for multi-dimensional querying and routing in wireless ad-hoc networks. In *P2P 2006: Sixth IEEE International Conference on Peer-To-Peer Computing*, 2006.

[20] G. Moro, G. Monti, and A.M. Ouksel. Merging G-Grid P2P systems while preserving their autonomy. In *P2PKM*, 2004.

[21] G. Moro, G. Monti, and A.M. Ouksel. Routing and localization services in self-organizing wireless ad-hoc and sensor networks using virtual coordinates. In *ICPS'06: IEEE International Conference on Pervasive Services 2006*, 2006.

[22] G. Moro, A.M. Ouksel, and W. Litwin. GGF: A generalized grid file for distributed environments. Technical report, DEIS Univ. of Bologna, Univ. of Illinois at Chicago, 2002.

[23] T. Moscibroda, R. O'Dell, M. Wattenhofer, and R. Wattenhofer. Virtual coordinates for ad hoc and sensor networks. In *DIALM-POMC '04: Proceedings of the 2004 joint workshop on Foundations of mobile computing*, pages 8–16, New York, NY, USA, 2004. ACM Press.

[24] S. Murthy and J. J. Garcia-Luna-Aceves. An efficient routing protocol for wireless networks. *Mob. Netw. Appl.*, 1(2):183–197, 1996.

[25] A.M. Ouksel and G. Moro. G-Grid: A class of scalable and self-organizing data structures for multi-dimensional querying and content routing in P2P networks. In *Proceedings of the Second International Workshop on Agents and Peer-to-Peer Computing, Melbourne, Australia, July 2003*, volume 2872, pages 123–137. Springer, 2003.

[26] M. A. Ouksel and O. Mayer. A robust and efficient spatial data structure: the nested interpolation-based grid file. *Acta Inf.*, 29(4):335–373, 1992.

[27] C.E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *SIGCOMM '94: Proceedings of the conference on Communications architectures, protocols and applications*, pages 234–244, New York, NY, USA, 1994. ACM Press.

[28] C.E. Perkins and E.M. Royer. Ad-hoc on-demand distance vector routing. In *WMCSA '99: Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, page 90. IEEE Computer Society, 1999.

[29] A. Rao, C. Papadimitriou, S. Shenker, and I. Stoica. Geographic routing without location information. In *MobiCom '03: Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 96–108. ACM Press, 2003.

[30] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.

[31] S. Ratnasamy, B. Karp, S. Shenker, D. Estrin, R. Govindan, L. Yin, and F. Yu. Data-centric storage in sensornets with ght, a geographic hash table. *Mob. Netw. Appl.*, 8(4):427–442, 2003.

[32] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–340, 2001.

[33] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M. Frans Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[34] Andrew S. Tanenbaum. *Computer networks: 3rd edition.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[35] L. Xiao and A.M. Ouksel. Tolerance of localization imprecision in efficiently managing mobile sensor databases. In *MobiDE '05: Proceedings of the 4th ACM international workshop on Data engineering for wireless and mobile access*, pages 25–32, New York, NY, USA, 2005. ACM Press.

[36] F. Ye, H. Luo, J. Cheng, S. Lu, and L. Zhang. A two-tier data dissemination model for large-scale wireless sensor networks. In *MobiCom '02: Proc. of the 8th annual international conference on Mobile computing and networking*, pages 148–159, New York, NY, USA, 2002. ACM Press.

[37] Wei Ye, John Heidemann, and Deborah Estrin. An energy-efficient mac protocol for wireless sensor networks. In *IEEE Infocom, 2002*, 2002.

[38] Wei Ye, John Heidemann, and Deborah Estrin. An adaptive energy-efficient mac protocol for wireless sensor networks. In *SenSys'03*, 2003.