Faculdade de Engenharia da Universidade do Porto

Overlay Networks for Intelligent Transportation Systems

Ricardo Filipe Pinho Lopes

Tese submetida no âmbito do

Mestrado Integrado em Engenharia Electrotécnica e de Computadores

Major de Telecomunicações

Supervisor: Ricardo Morla

July 2008

U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES 2007/2008

A Dissertação intitulada

"Overlay Networks for Intelligent Transportation Systems"

foi aprovada em provas realizadas 15/Julho/2008

o júri

Presidente Professor Doutor Manuel Alberto Pereira Ricardo
Professor Associado da Faculdade de Engenharia da Universidade do Porto

[illegible]

Professor Doutor Carlos Miguel Ferraz Baquero Moreno
Professor Auxiliar da Escola de Engenharia da Universidade do Minho

[illegible]

Professor Doutor Ricardo Santos Morla
Professor Auxiliar da Faculdade de Engenharia da Universidade do Porto

Ricardo Santos Morla

O autor declara que a presente dissertação (ou relatório de projecto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extractos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são correctamente citados.

Autor – Ricardo Filipe Pinho Lopes

Ricardo Filipe Pinho Lopes

Faculdade de Engenharia da Universidade do Porto

Abstract

The current research in Intelligent Transportation Systems (ITS) is lacking adequate tools to create overlay networks and services applied to ITS. Developing a custom made overlay network for ITS is complex and developers may focus on less important implementation details rather than on the algorithms and scenarios suitable for ITS overlays. In this report we present a new framework that provides software developers with tools that can aid them in the development and implementation of overlay networks for ITS, the ITS Overlay Framework. Its main contributions are: providing an approach to the development and simulation of ITS overlays that eases the transition to testbeds and real-case usage; allowing users to create custom vehicle groupings based on coordinates, each associated with a different overlay network; allows users to define how nodes join a group overlay network based on their position, update themselves in the group overlay or switch to another group overlay, and send messages between themselves in the same group; supporting the development of different overlay network topologies and protocols. We have used the ITS Overlay Framework to implement and run three test scenarios that explore different groups and different overlay network topologies.

"Science never solves a problem without creating ten more."

George Bernard Shaw

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

An Intelligent Transportation System (ITS) is a system that aims to improve transportation safety and mobility and to enhance productivity through the use of advanced communications technologies. ITS covers a wide range of wired and wireless communications and electronics technologies. When these technologies are integrated into the transportation system's infrastructure, and into vehicles themselves, they help preventing congestion, improving safety, and enhancing productivity.

In ITS, communication and sensor technologies are used to help improve transportation conditions e.g. in heavy traffic freeways. Given for example GPS receivers, wireless communications and overlay networks may be applied in ITS and help improve transportation conditions. ITS uses sensor technologies such as road sensors, radars, and traffic monitoring and information systems.

An overlay network is a computer network built on top of another network. Nodes in the overlay can be thought of as being connected by virtual links or tunnels. Each of these links corresponds to a path possibly through different physical links in the underlying network. Overlay packets are exposed only at tunnel endpoints. Overlays may thus support different routing and addressing protocols from the base network.

A relevant example of a kind of overlay network is P2P [2]. Most P2P networks are overlay networks over IP, the Internet Protocol. A P2P overlay network can be of two types: structured and unstructured. Structured P2P networks are overlay networks based on Distributed Hash Tables (DHT) and unique keys, where nodes (also called peers) are placed in predefined locations. In unstructured P2P networks, peers are placed without knowledge about the network topology. Typically, this means that peers need to flood the network with queries to know about other peers. Overlay networks can also support geographic routing [5, 6, 7] between peers. This means that data is forwarded based on a geographical destination instead of e.g. an IP address.

1.2 Goals

The main goal of this project is to create a framework that enables developers and researchers to implement overlay networks over IP for intelligent transportation systems (ITS) [8] where the mobile nodes position themselves in the overlay network based on vehicle locations and groups of vehicles using GPS or traffic simulation data. The created overlay networks through the framework may take advantage of access network technologies such as wireless networks 802.11, ad-hoc networks and third generation mobile radio networks [9].

In particular the framework would allow users to:

- Create different types of vehicle groupings defined by the user e.g. vehicles with the same speed, within a range of 1Km. In order to create groups of vehicles in the overlay networks, models of traffic flows [10, 8] must be understood.
- Provide means to develop a mechanism for creating and establishing overlay networks based on GPS localization [5, 6, 7]. The framework should enable the use of different overlay network topologies defined by the user e.g. centralized topology.
- Use the previous mechanism to support updating a node in an overlay network. We expect this kind of network will be very volatile and capable of changing the entire network with minimal delay. This is needed because vehicles travel at different speeds and directions.
- The framework should also support exchanging ITS related messages between nodes e.g. related to road safety such as sudden braking or crashes.
- Validate the developed solution with a help of traffic simulation data.

1.3 Structure

Chapter two is the **Related Work** and it contains a state-of-art study in the following fields of research: Intelligent Transportation Systems (ITS), Overlay Networks, Traffic Algorithms and Simulators and Software Development Frameworks. This study provides a background for the main subjects of this thesis, helping in the development of our framework.

In chapter three we present our solution, an **ITS Overlay Framework**. We describe its elements: the **Overlay**, the **Group Server** and the **Simulator**. We also present and explain the framework's architecture in a layer scheme with three layers: the Network Communications Layer, the Management Layer and the Application-Level Layer.

The next chapter, chapter four, is where we present our experiments and tests of the framework. We begin to present the default implementations to use with the framework and how to write an application (Overlay Node Client, Group Server or Simulator) using the framework. We also present three test scenarios using the framework that provide relevant results for the framework evaluation. And we have a final analysis on what the framework supports and does not.

The final chapter presents the final conclusions of this thesis and future work.

Chapter 2

Related Work

2.1 Intelligent Transportation System (ITS)

An Intelligent Transportation System (ITS) aims to improve transportation safety and mobility and to enhance productivity through the use of advanced technologies [11]. According to the Research and Innovative Technology Administration (RITA), ITS have sixteen types of technology based systems which are divided into two types of systems: intelligent infrastructure systems and intelligent vehicle systems.

2.1.1 Technologies Used in ITS

These systems are supported by numerous technologies (e.g. electronics technologies, communication technologies, sensor technologies).

The impact of electronics technologies [12] on the automobile industry has grown over the years. Microelectronics can be applied in ITS as it contributes significantly to the performance, efficiency and safety of automobiles and helps improving traffic control. An important example of the use of this kind of technologies in ITS is the vehicle location and navigation systems such as GPS systems that make use of the newest electronics technologies in order to be more efficient.

There are also numerous sensor technologies that can be used in an ITS such as road sensors and radars in order to monitor the traffic. Inter-vehicle range detection has become one of the key methods to avoid collisions and ensure safety to the vehicles. Therefore, automotive radar is expected to be a key technology in ensuring driving safety in the transportation systems. One example of this type of technologies is a Chaos Radar for Collision Detection and Vehicular Ranging [13].

ITS can take advantage of various network technologies. A relevant network technology for this purpose is the third generation mobile radio network [9]. The research in the field of communications for ITS presents third generation mobile radio networks in the possible provision of services for ITS. Mobile phone location determination is being studied [14] as a complement to other location-determination systems.

User Services Bundle	**User Services**
Travel and Transportation Management	En-Route Driver Information Route Guidance Traveler Services Information Traffic Control Incident Management Emissions Testing and Mitigation Demand Management and Operations Pre-trip Travel Information Ride Matching and Reservation Highway Rail Intersection
Public Transportation Operations	Public Transportation Management En-Route Transit Information Personalized Public Transit Public Travel Security
Electronic Payment	Electronic Payment Services
Commercial Vehicle Operations	Commercial Vehicle Electronic Clearance Automated Roadside Safety Inspection On-board Safety Monitoring Commercial Vehicle Administration Processes Hazardous Materials Incident Response Freight Mobility
Emergency Management	Emergency Notification and Personal Security Emergency Vehicle Management
Advanced Vehicle Control and Safety Systems	Longitudinal Collision Avoidance Lateral Collision Avoidance Intersection Collision Avoidance Vision Enhancement for Crash Avoidance Safety Readiness Pre-Crash Restraint Deployment Automated Highway System
Information Management	Archived Data

Table 2.1: User Services

2.1.2 ITS User Services

ITS technologies can be used in various interrelated user services for applications to solving transportation problems. According to ITS Executive Summaries prepared by Lockheed Martin Federal Systems, Odetics Intelligent Transportation Systems Division [1], to date, thirty-one user services have been identified. This list of user services is not final. The user services can be inserted into seven categories as shown in Table 2.1.

2.1.3 ITS Architecture

ITS bring a great variety of options regarding transportation needs. In order to invest resources to develop system solutions that are compatible worldwide, a common ITS architecture structure should provide overall guidance to ensure system, product, and service compatibility/interoperability. To this purpose the U.S. Department of Transportation presents a National ITS Architecture [15, 1].

The National ITS Architecture defines a framework in which multiple design approaches can be developed, each one to specifically meet the individual needs of the user, while maintaining the benefits of a common architecture. The architecture defines the functions (e.g. gather traffic information or request a route) that must be used in order

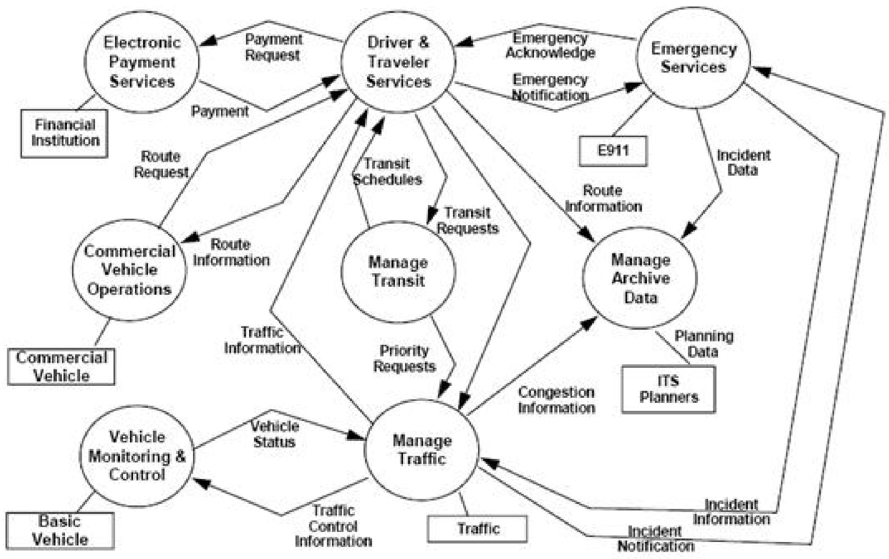

Figure 2.1: Simplified Top Level Logical Architecture [1]

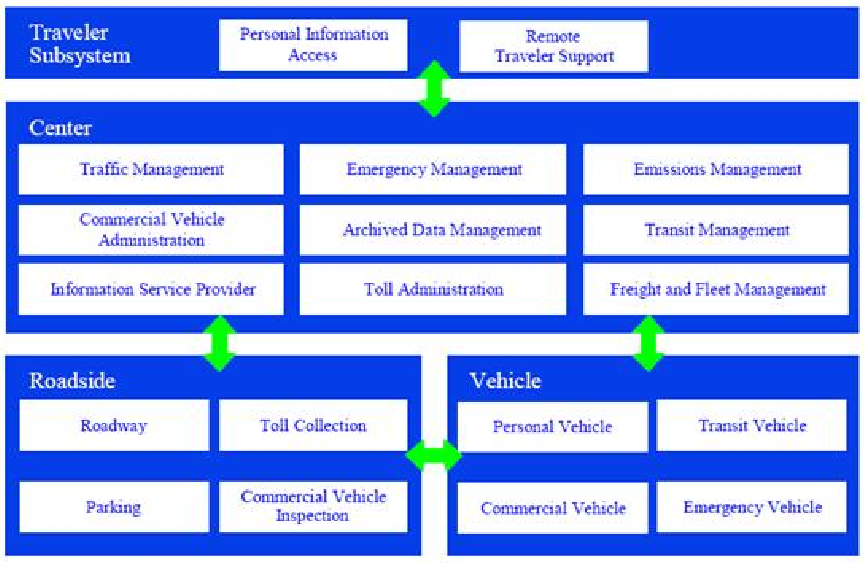

Figure 2.2: Architecture Systems and Subsystems [1]

to implement a user service, the subsystems where these functions reside (e.g. roadside or vehicles), the interfaces/information flows between the physical subsystems and the communications requirements for the information flows (e.g. wired or wireless). It also identifies and specifies product standards and the requirements needed in order to support national interoperability. A simplified top level logical architecture and the architecture systems and subsystems of the National ITS Architecture are represented in Figure 2.1 and Figure 2.2.

Besides the American National ITS Architecture there are also some other relevant ITS frameworks such as the European ITS Framework KAREN or FRAME [16] or even the iTransIT ITS Framework [17].

2.1.4 Security in ITS

Since ITS is based on information technologies and encompasses a wide range of information (e.g. traffic control, safety, financial, and personal), it is necessary to guaranty information security against various attacks.

According to the results from an information security analysis that was based on the National ITS Architecture [18], information security requirements were not thoroughly considered in the National ITS Architecture. Currently there is neither a Security Architecture nor a Security Policy for ITS. It also refers that ITS designs should include measures to protect against a wide range of security threats and should contain security services and infrastructures integrated into the overall system design to provide adequate security.

2.2 Overlay Networks

Widely-distributed systems need to track its nodes and be able to send messages among those nodes. This capability is often called an overlay network [19], because it provides an application with customized network functionality that runs as layer over the IP networking. So an overlay network is a virtual network formed by nodes cooperating between themselves that share the same underlying network (IP). The overlay network can have a completely different topology and different protocols for naming and routing.

Overlay networks are widely used and maybe the most relevant example of this kind of networks are Peer-to-Peer (P2P) network overlays as they provide services for large-scale data sharing, content distribution and application-level multicast applications.

2.2.1 Distributed Systems Topologies

The development of peer-to-peer overlay networks has renewed interest in decentralized systems [20, 21]. The Internet itself is the largest decentralized system in the world. But ironically in the 1990s many systems built on the Internet were completely centralized.

The growth of the Web meant most systems were single web servers running in expensive facilities. A topology of a distributed system is how the different nodes in the system organize themselves. There are numerous topologies for distributed systems. The basic topologies used on the Internet include: centralized, decentralized, hierarchical and ring systems, and combinations of these ones creating hybrid systems. In order to better understand overlay networks topologies we describe some topologies of distributed systems:

2.2.1.1 Centralized Topology

Centralized systems are the most common form of topology. They can be typically seen as the client/server model used by databases, web servers, and other simple distributed systems (see Figure 2.3). There is one intelligent terminal that contains all information and function designated as server with many clients connecting directly to the server to send and receive information.

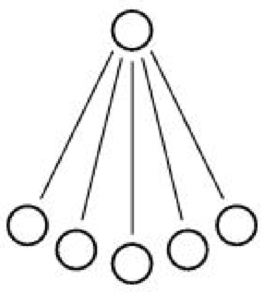

Figure 2.3: Centralized Topology

2.2.1.2 Ring Topology

When there are many clients, a single centralized server cannot handle all the clients, so a frequent solution is to use a group of machines arranged in a ring to act as a distributed server (see Figure 2.4). This is typically done to provide failover and load-balancing capabilities to the distributed server.

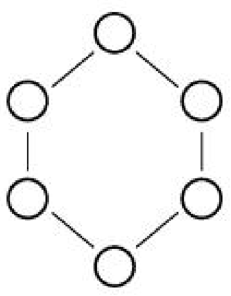

Figure 2.4: Ring Topology

2.2.1.3 Hierarchical Topology

Hierarchical systems have long been used on the Internet, but in practice are often overlooked as a distinct distributed systems topology (see Figure 2.5). For example, the best known hierarchical system on the Internet is the Domain Name Service (DNS), where authority is delegated from the root name-servers to the server for the registered name and often down to third-level servers. The Network Time Protocol (NTP) uses another hierarchical system.

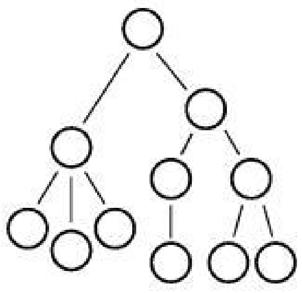

Figure 2.5: Hierarchical Topology

2.2.1.4 Decentralized Topology

The final basic topology used in the Internet we consider is that of fully decentralized systems [21, 22], where all peers communicate symmetrically and have equal roles (see Figure 2.6). Many file-sharing systems are designed to be decentralized, such as Gnutella [23, 24] or Freenet [25]. The Internet routing architecture itself is largely decentralized, with the Border Gateway Protocol used to manage the peering links between various autonomous systems.

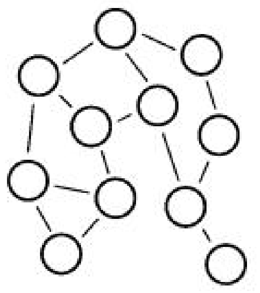

Figure 2.6: Decentralized Topology

2.2.1.5 Hybrid Topology

Distributed systems can have a more complex topology than the basic ones referred above. These systems often combine several basic topologies into one system, creating a hybrid topology. In these system nodes usually have multiple roles e.g. a node can be part of an hierarchy in a part of the system while having a centralized interaction with another part of the system.

2.2.2 Peer-to-Peer Overlay Networks

P2P overlay networks can be seen as fully-distributed, cooperative network design with peers building a self-organizing system. With the evolution of Internet, P2P has taken an important part in data sharing and content distribution. Because of this crescent use of P2P networks and the number of P2P networks schemes is also increasing [2]. In Fig-

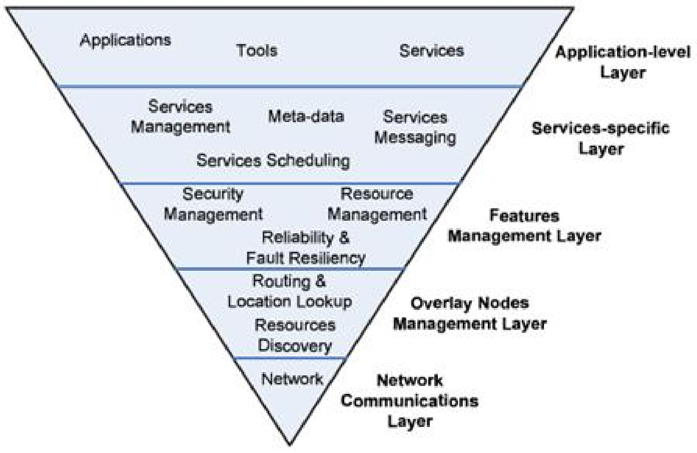

Figure 2.7: P2P Overlay Architecture [2]

ure 2.7 we can see an abstract P2P overlay architecture, illustrating the components in the overlay communications framework. In the lower layer, the Network Communications layer, the networks characteristics of machines connected over the Internet are described. The Overlay Nodes Management layer takes care of the management of peers, which include routing and addressing of peers. The security and reliability of the overlay network is covered by the Features Management layer which is responsible for assuring the robustness of P2P systems. The underlying P2P infrastructure and the application-specific components are supported by the Services Specific layer through scheduling of parallel and computation-intensive tasks, content and file management. On the Application-Level layer of P2P overlay networks we can find applications, tools and services that are implemented above the underlying P2P overlay network infrastructure.

Based on this architecture we can distinct two classes of P2P networks who have different characteristics: Structured and Unstructured P2P networks.

DHT	DOLR	CAST
put (key, data)	*publish (objectID)*	*join (groupID)*
remove (key)	*unpublish (objectID)*	*leave (groupID)*
value = get (key)	*sendToObj (msg, objectID, [n])*	*multicast (msg, groupID)* *anycast (msg, groupID)*

Table 2.2: Operations of Each Service

2.2.3 Structured P2P Overlay Networks

The network topology of structured P2P overlay networks [2, 26] is firmly controlled and content is placed at specific locations instead of random peers thus enabling more efficient queries. Structured overlays use services such as distributed hash tables, scalable group multicast and anycast, and decentralized object location and routing [21]. These services sustain new classes of highly scalable, resilient, distributed applications, including cooperative archival storage, cooperative content distribution and messaging. The operations of each service are resumed in Table 2.2.

The Distributed Hash Table (DHT) service provides the same functionality as the traditional hashtable, by storing the mapping between a key and a value. On DHT-based systems uniform random NodeIDs are assigned to the set of peers into a large space of identifiers. From the same identifier space, unique identifiers called keys are assigned to data objects (values). The overlay network protocol is in charge of map the keys to a unique live peer in the overlay network. The interface of DHT implements a simple store and retrieve functionality of $key, value$ pairs, where the value is always stored at the live overlay node with its key mapped by the overlay network protocol. These values can be of any type. This behavior is illustrated in Figure 2.8. Using a key, a store operation ($put(key, value)$) and a retrieval operation ($value = get(key)$) can be invoked to store and get the data object corresponding to the key involving routing requests to the peer with the assign key.

There are different DHT-based systems with different organization schemes for the data objects and its key space and routing strategies but in theory, DHT-based systems can

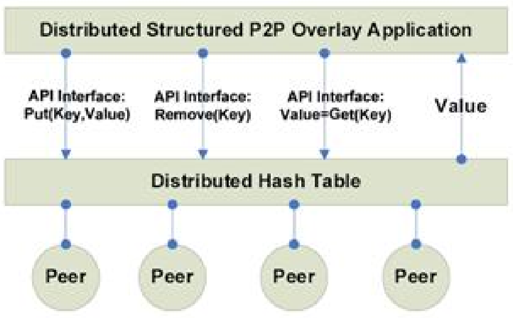
Distributed Structured P2P Overlay Application
API Interface: Put(Key,Value)
API Interface: Remove(Key)
API Interface: Value=Get(Key)
Value
Distributed Hash Table
Peer
Peer
Peer
Peer

Figure 2.8: DHT in Structured P2P Overlay [2]

assure that any data object (value) can be found by exchanging a small number messages: $O(logN)$ messages on average, where N is the number of nodes in the overlay.

The Decentralized Object Location and Routing (DOLR) provides a decentralized directory service where each endpoint (object) has an objectID and may be placed anywhere within the system. The presence of endpoints is announced by applications, by publishing their locations. This way, a client message with a particular objected as the destination will be delivered to a near endpoint with this name. The maintenance of the distributed directory during changes in the underlying nodes or links is part of this process.

The group anycast and multicast (CAST) service provides scalable group communication and coordination. The nodes may join and leave a group, multicast messages to the group, or anycast a message to a member of the group. The group is represented as a tree and because of that membership management is decentralized. Therefore CAST can support large and highly dynamic groups. Moreover, if the overlay network protocol is proximity aware, then multicast is efficient and anycast messages are delivered to a group member near the anycast sender.

The DOLR and CAST abstractions are similar in some aspects: both of them maintain sets of decentralized endpoints by their proximity in the network, using a tree of routes from the endpoints to a common root associated with the set of endpoints. However, while DOLR is more focused towards object location, CAST is more associated with group communication. Although they use the same basic mechanism their implementations combine different policies. On the other hand, DHT provides a very different service as it provides a scalable repository for $key, value$ pairs.

Now we will review some of the most significant Structured P2P overlay networks (CAN [27], Chord [28], Pastry [20] and Tapestry [29]) and although they do not make up an exhaustive list of structured overlays, they represent a cross-section of existing systems. All of the reviewed systems use DHT functionality on an Internet-like scale as they are largely decentralized distributed systems.

1. CAN

   CAN [27] is a Structured P2P overlay network designed to be scalable, self-organizing and fault-tolerant. The overlay system architecture of CAN is a virtual multi-dimensional ID coordinate space. The coordinate space entirely logical as it is dynamically partitioned among a N number of peers in the system. This way each pear possesses its individual zone within the coordinate space. CAN adopted a lookup query protocol that uses key, value pairs to map a point P in the coordinate space using uniform hash function. This system has two parameters: N the number of peers in the network and d number of dimensions. The CAN has a routing performance of $O(d * N^{1/d})$ and a routing state of $2 * d$. When a peer joins the system, it needs its own zone in the coordinate space. This is resolved by splitting existing peer's zone in half. When a peer leaves the network, an instantaneous takeover algorithm ensures that a neighbor of the failed peer takes over its zone and starts a takeover timer. The neighbor set of the peer that takes over the zone is updated by eliminating the peers that are no longer its neighbors. Then every peer in the system sends soft-state updates to guarantee that all of their neighbors will learn about the change and update their own neighbor sets. The number of neighbors a peer maintains depends only on the number of dimensions of the coordinate space (i.e. $2 * d$) and it is completely independent of the total number of peers in the system. When subjected to faults of peers the system will not cause network-wide failure since multiple peers are responsible for each data item. On failure application retries.

2. Chord

   Chord [28] assigns keys to its peers using consistent hashing [30]. Chord's architecture design uses uni-directional and circular NodeID spaces. Hash functions assign peers and data-keys an m-bit identifier chosen by hashing the peer's IP address, while a key identifier (k) is produced by hashing the data key. The number of bits of the identifier must be large enough to guarantee that the probability of keys hashing to the same identifier is insignificant. The identifiers are ordered on an identifier circle modulo $2m$. The first peer whose identifier is equal to or follows k in the identifier space is assigned with key k thus been called the successor peer of key k, denoted by $successor(k)$. The $successor(k)$ is the first peer clockwise from k in the identifier circle. This identifier circle is named Chord ring. When peer n leaves the Chord system, all of its assigned keys are reassigned to peer n's successor. This way, peers join and leave the system with $(logN)^2$ performance. The lookup query protocol adapted by this system consists of matching a key with a NodeID. Given the N the total number of peers in the system, the routing performance is only $O(logN)$. This may be efficient but the performance degrades as the routing state information of each peer gets out-of-date. The routing state and scalability of this overlay system is $logN$. Like in CAN, failure of peers will not cause network-wide failure and on failures, application retries. Replicate data on multiple consecutive peers.

3. Tapestry

   Both Pastry and Tapestry [29] employ decentralized randomness to achieve both load distribution and routing locality. The main difference between the two systems is the management of network locality and data object replication. Tapestry's architecture uses a Plaxton-style global mesh network [31] where peers can take on the roles of servers (where data objects are stored), router (forward messages) and clients (make requests). Each peer has local routing maps to incrementally route overlay messages to the destination ID digit by digit. These local routing maps have multiple levels where each of them represents a matching suffix up to a digit position in the ID space. When a message reaches peer n, this peer shares a suffix of at least length n with the destination ID. In order to locate the next router, the next level map $(n + 1)$ is examined to locate the entry match of the value of the next digit in the destination ID. In a system with N peers using NodeIDs of base B, using this routing method guarantees that any existing peer in the system can be located within at most $log_B N$ logical hops. "Since the peer's local routing map assumes that the preceding digits all match the current peer's suffix, the peer needs only to keep a small constant size (B) entry at each route level, yielding a routing map of fixed constant size $B * log_B N$" [2]. Like in the other systems the failure of peers will not cause network-wide failure. To assure the robustness of the system when subjected to faults, the overlay replicates data across multiple peers and keeps track of multiple paths to each peer.

4. Pastry

The Pastry's [20] architecture is very similar to Taspertry's architecture since it is based on Plaxton-style global mesh network. It makes use of Plaxton-like prefix routing, to build self-organizing decentralized overlay network, where each peer redirects client requests and interacts with local instances of one or more application. In Pastry each peer is assigned with a NodeID ($128 - bit$ peer identifier). This NodeID is assign randomly to a peer when it joins the system and is used to locate the peer in a circular NodeID space which goes from 0 to 2^{128-1}. In a network with N peers, Pastry routes to the numerically closest peer to a given key in less than $log_B N$ steps under normal operation (where $B = 2^b$ is a configuration parameter with typical value of $b = 4$). The keys and NodeIDs are considered a sequence of digits with base B. Messages are routed to the peer whose NodeID is numerically closest to the given key. A message is forwarded to a peer whose NodeID shares a prefix with the key. This prefix has to be a least one digit longer than the prefix the key shares with the current peer NodeID. A Pastry peer maintains a routing table, a neighborhood set and a leaf set. The peer routing table has $log_B N$ rows each row with $B - 1$ entries. Each of these entries contains the IP address of peers whose NodeID have the appropriate prefix and it is chosen based on close proximity metric. "The choice of b involves a trade-off between the size of the populated portion of the routing table (approx:$(log_B N) * (B - 1)$ entries) and maximum number of hops required to route between any pair of peers $(log_B N)$" [2]. As for the failures and the robustness of the system, Pastry has identical mechanisms to Tapestry.

2.2.4 Unstructured P2P Overlay Networks

Before the appearing of structured peer-to-peer (P2P) overlay networks, some applications used a cooperative application model to allow participants to access each others' resources for mutual benefit. This type of systems was designated as Unstructured P2P overlay networks and was popularized by file-sharing applications such as Gnutella. An Unstructured P2P system is a system or protocol where the nodes perform actions (such as routing messages) for each other, where no rules exist to define or constrain connectivity between nodes. Messages between endpoints can take arbitrary paths through the system. Because there is no structure in these types of overlay networks, in order to maintain full connectivity between nodes it generally means that each node must maintain routing information for all possible destinations in the network. The fact that this routing information grows linearly with the size of the network is clearly a factor in limiting the scale of these networks.

In Unstructured P2P overlay networks, the overlay network arrange peers in a random graph in flat or hierarchical manners and normally use flooding on the graph to query content stored by overlay peers. Each peer that receives a query will evaluate it locally on its own content, and will support complex queries. Queries for content that are not widely spread between the peers must be sent to a large fraction of peers and there is no combination between topology and data items' location thus turning this system inefficient. Now we shall review some of the more influential Unstructured P2P overlay networks: Gnutella [23, 24, 22] and Freenet [25, 22].

1. Gnutella

   Gnutella is a decentralized peer-to-peer system, consisting of hosts connected to one another over TCP/IP running software that implements the Gnutella protocol. This connection of nodes forms a network of computers exchanging Gnutella messages: queries, replies to queries, and other control messages used to discover nodes flooding the network with messages (see Figure 2.9). This P2P overlay network allows its nodes to share resources between themselves. The most recent implementations are almost exclusively focused on data files so that any host can offer local files to other hosts to download. The nodes of the network are controlled by users running the application software. The users can participate in the network by: specifying a list of local files to share within the network, searching for files shared by other users in the network and downloading files shared by other users.

   When a node wants to participate in the network it must join the Gnutella network by connecting with an existing node in the network. In the current implementation of Gnutella, new nodes are allowed to easily participate in the network just by connecting to a random node's IP address provided via DNS or script running on a web site by mechanisms known as "host caches". In order to reach more total Gnutella nodes a node that joins the network typically connects to multiple existing nodes in the Gnutella network. Once a node has connected to the network, it communicates with its neighboring nodes using Gnutella protocol messages and accepts incoming

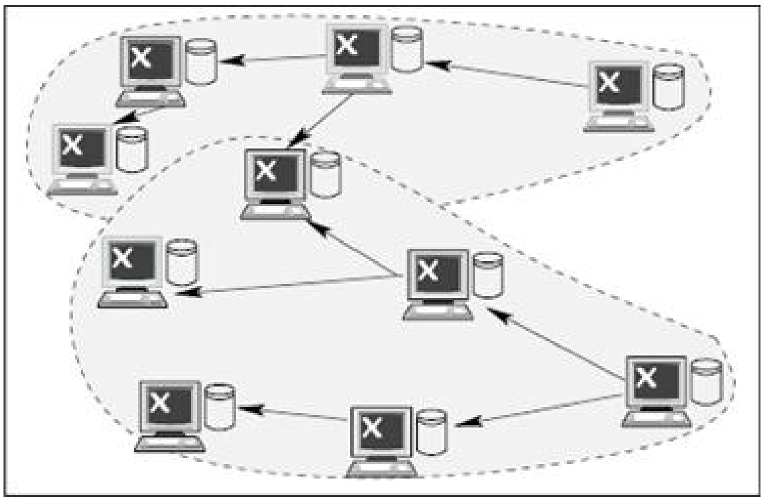

Figure 2.9: TTL Scopped Flooding in Gnutella

connections from new nodes that wish to join the network. The main Gnutella protocol messages are described in the table 2.3.

2. Freenet

   Freenet and Gnutella have a very similar topology but in Freenet the main purpose is to build a secure global information storage system. Existing nodes in Freenet networks route messages through the network and also store some data that do not belong to them thus creating a very large, distributed file storage system with plenty of redundancy. Since many nodes share the same files, the network is quite fault-tolerant. When a new node is connecting to the network it needs to know about existing nodes in order to successfully connect. Clients must find the first node on their own using a host cache or knowledge from a friend, because there is no central authority. The user with the Freenet software can participate with the network by: specifying disk storage space that will be used to store some network data that do not belong to them, sending an "insert" message to add new files to the network and sending a "request" message with a key to get a file from the network. These steps are quite different from the steps to participate in a Gnutella network, because a user cannot access any of the encrypted data stored locally nor can request files from specific nodes. When a user needs to request content from the network

Type	**Description**
Ping	A request for information about other node(s)
Pong	A reply carrying information about a node (e.g. number of files shared)
Push	A mechanism that allows a firewalled node to share data
Query	A request for a resource (e.g. searching for a file)
Query Hit	A response identifying an available resource (e.g. a matching file)

Table 2.3: Gnutella Protocol Messages (Bye excluded)

they must send a request to the network and wait until a node returns the desired contents. Additionally users cannot search the network for arbitrary files; the users must already know the key that describes the desired file.

Like it was said before the main purpose of the Freenet architecture is to create a secure network. This can be done because each node in the network only knows its closest neighbors. This way it is impossible for a node to discover in which node in all the network the desired data is stored. In this respect, the network is very resistant to tampering and censorship. For each file in the network a unique identifying hash can be generated, but the hash itself do not allow to determine what data it represents. Since nodes route and store these hashes, they do not know the data content represented by the hashes. Furthermore, each node has stored data encrypted with a key that is not available to them. Thus, each node does not know the nature of the content they are storing for the network.

2.2.5 Geographic Overlay Networks

While routing and addressing are made with keys based on a Distributed Hash Table (DTH) in the Structured P2P overlay networks and made by flooding the network with queries in the Unstructured P2P overlay networks, geographic overlay networks' (GOnet) routing and addressing is made based on the physical location of an host [5]. Therefore a GOnet would enable location-based, proximity-based and special application classes. In a GONet over IP routing and addressing can be made using GPS data [6, 7].

The location-based applications depend of the host location, e.g., reading only sensors at particular map positions and tracking. Proximity-based applications depend upon discovery of nearby hosts, e.g., to find resources such as printers.

A GONet not only provides support for geographic addressing and routing but also enables geographic overlays to be incrementally deployed on the current Internet with routing tables that are tightly delimited and small. The ability to recursively build overlays on overlays allows creation of a routing hierarchy in a GONet to achieve typical hop counts that are log proportional to network size.

The nodes in GONet depend on the base network for link connectivity. For this reason a path to a geographically nearby neighbor may take a large number of hops in the underlying network thus lowering the efficiency. In spite of the low efficiency, the desired geographic network topography is produced, which allows unicast geographic routing and spatialcasting.

The GONet introduces the use of tunnels to simplify the methods to discover physically close neighbors. This way source routes need not to be discovered and only edge endpoints need to be known thus maintaining the route between endpoints is responsibility of the underlying network.

2.2.6 Applications and Tools

The following applications can be used for designing and deploying overlay networks, therefore we can take advantage of these applications in the development of an overlay.

2.2.6.1 X-BONE Overlay System

The X-Bone [32] provides a web-based GUI and back-end control software to deploy Internet overlays. It is a free and open source software implemented in Perl available as a FreeBSD port (3.3+, 4.x+), a Linux RedHat RPM (6.x+), and as a source tar file. It can be used for networking research to test protocols and apps on new topologies.

The X-Bone is known as a general mechanism for network virtualization. An X-Bone overlay includes both end hosts and routers in the overlay, and includes support for overlays at all points in the virtual network.

2.2.6.2 P2: Declarative Network

P2 [33] is a system used to specify and execute overlay networks in a highly compact and reusable form making use of a high-level declarative language. It can automatically compile high-level specifications to a dataflow-oriented runtime system, which can be used itself by expert programmers to specify efficient overlays.

2.3 Algorithms and Simulation of Traffic Flow

”Traffic-flow characteristics such as flow, density, and space mean speed (SMS) are critical to Intelligent Transportation Systems (ITS)” [10]. Therefore it is necessary to identify traffic flow characteristics for application in the ITS. Such characteristics can be determined by direct and indirect methods [10]. Since in this project there is a need to identify vehicles for their GPS data (location, velocity and direction) traffic simulation is needed to test the system. In order to define how vehicles can be grouped results of simulations of freeway and metropolitan traffic flow are important [8].

The use of traffic simulation can bring benefits e.g. short term forecasting (to determine actions following an incident that changes the roadway), anticipatory guidance for Advanced Traveler Information Systems (ATIS) to help drivers make better decisions, determine how to improve a transportation infrastructure, and planning for road blocks, closures and construction sites.

There are several modeling approaches (e.g. microscopic, macroscopic and mesoscopic simulation models) that cover different situations (e.g. freeways, intersections) each one with a different level of detail [34].

Macroscopic traffic simulation also called continuous flow simulation is mainly used in traffic flow analysis. In the macroscopic model, observed statistical patterns of traffic flow are explicitly imposed collectively on the vehicles in the simulation. Using this system level approach to govern the behavior of individual element yields somewhat limited results as such simulations can only simulate what is already known.

On the other hand, on microscopic traffic simulations [8], every vehicle is simulated. In this kind of simulation models, vehicle movements are characterized by three behaviors: accelerations, braking decelerations and lane changes. In order to achieve accuracy in modeling the traffic, many factors must be considered leading to a simulation model with a large number of parameters (50 parameter models are common).

Mesoscopic models are a more recent type of simulation models that try to combine the advantages of microsimulation (detail) and macrosimulation (scability to larger networks). These models normally describe the traffic entities at a high level of detail, but their behavior and interactions are described at a lower level of detail.

2.3.1 Traffic Simulators

2.3.1.1 Microsimulation of road traffic

This is an online microscopic simulator of road traffic developed in JAVA [35] with its source code available. This simulator uses two traffic models: the Intelligent-Driver Model (IDM) to simulate the longitudinal dynamics, i.e., accelerations and braking decelerations of the drivers and the Lane-Change Model MOBIL where lane change takes place if potential new target lane is more attractive, i.e., the "incentive criterion" is satisfied and the change can be performed safely, i.e., the "safety criterion" is satisfied.

2.3.1.2 FreeSim: A Free Real-Time Freeway Traffic Simulator

FreeSim [36] is a fully-customizable macroscopic and microscopic free-flow traffic simulator that allows multiple freeway systems to be easily represented and loaded into the simulator and has its source code available for download.

In FreeSim the vehicles can communicate with the system that monitor the traffic on the freeways, which makes FreeSim ideal for Intelligent Transportation System simulation.

2.3.1.3 Simetron: Metropolitan Traffic Simulator

Simetron [37] is an open-source traffic simulator software for metropolitan networks. It provides an open plug-in framework that allows researchers and practitioners to integrate and test their own simulation models.

2.4 Software Development Frameworks

A software development framework is a re-usable design that provides tools to develop a customized application. A software development framework aids software developers by containing support programs, code libraries, or other software to help solving problems for a given domain and provides a simple API.

Frameworks are designed in order to facilitate software development by providing low level details and resources for a software domain. This way, software developers can spend more time on meeting the software requirements and specifications rather than caring about the low level code behind a software system, thus increasing productivity in software development.

2.4.1 PlanetSim

PlanetSim [3] is a new object oriented simulation / experimentation framework for large scale overlay networks and services with three main contributions:

- provides a unifying approach to simulation / experimentation that eases the transition from simulation to network testbeds such as PlanetLab;
- distinguishes clearly between the implementation of overlay algorithms like Chord and Pastry, and the applications and services built on top of them (DHT, CAST, DOLR, etc.);
- offers a layered and modular architecture with well defined hotspots documented using classical design patterns.

PlanetSim has been implemented in the Java language to reduce the learning curve of the framework. To validate the utility of the framework, it comes with two implemented overlays (Chord and Symphony) and a various services like CAST, DHT, and DOLR. PlanetSim reproduces the measures of these environments and is also efficient in its network implementation.

PlanetSim Architecture Layered Design

PlanetSim has a well-structured and modular architecture and makes use of the Common API for Structured Overlays [26]. Its architecture is divided in three main extension layers constructed one atop another: the Application Layer, the Overlay Layer and the Network Layer. We can see the architecture layered design in Figure 2.10

So the framework has the following three main extension points (hotspots), one for each layer:

- Application - The Application code can be used to send or route messages as well as access underlying node routing state. Any application created at this level can then be run or tested against any structured overlay in the next layer.
- Node - The Node code allows the development of overlay protocols or algorithms like Chord. At this level nodes exchange messages using Ids and IP Address plus Id.
- Network - The Network code provides the means to create customized Networks (RingNetwork, CircularNetwork, RandomNetwork).

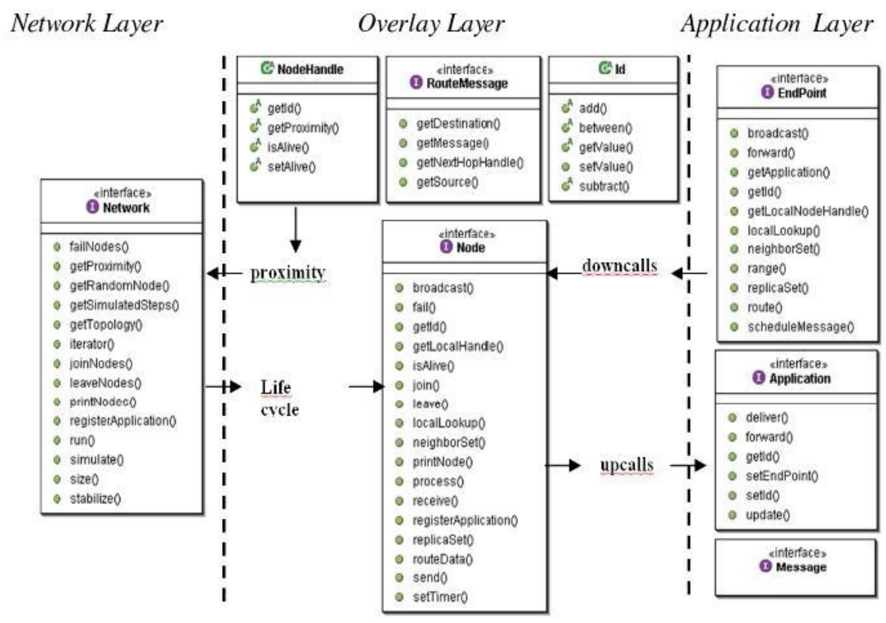

Figure 2.10: PlanetSim's Architecture Layer Design [3]

Because of this layered approach two main user roles have been identified: the users interested in overlay services and others focused on overlay infrastructures.

PlanetSim has been used to implement and evaluate overlay networks such as Chord and Symphony, and overlay services such as Scribe application level multicast, and keyword query systems over distributed hash tables.

2.4.2 OverSim

OverSim [4] is an open-source flexible overlay network simulation framework that uses the OMNeT++/OMNEST [38] simulation environment. Oversim includes several models for structured (e.g. Chord, Kademlia, Pastry) and unstructured (e.g. GIA) P2P protocols. The implemented protocol models can be used not only for simulation but also as real world networks.

In order to facilitate the implementation of additional more comparable protocols OverSim provides common functions like a generic lookup mechanism for structured peer-to-peer overlay networks.

OverSim Modular Architecture

The OverSim simulation framework was designed as a modular simulation framework with four modules. An overview of its architecture is illustrated in Figure 2.11.

- Simulation framework OMNeT++

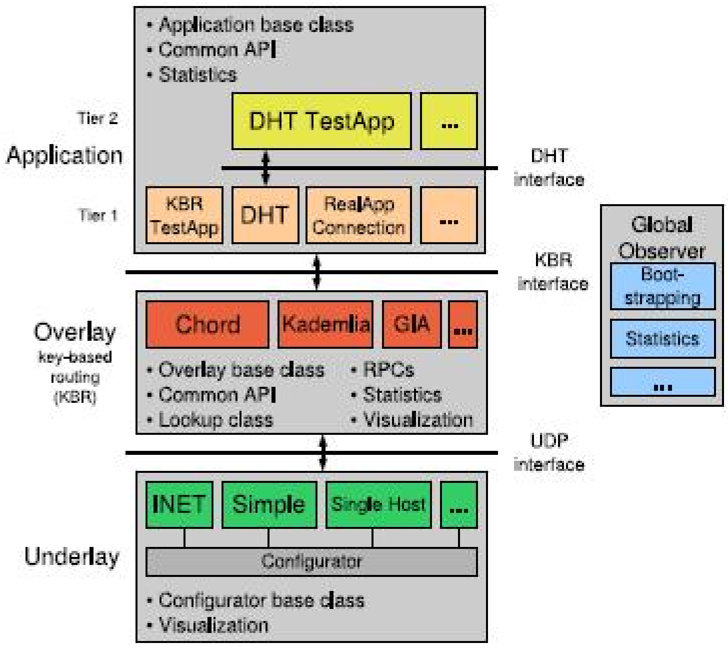

Figure 2.11: Modular Architecture of OverSim [4]

OverSim uses discrete event simulation (DES) to simulate exchange and processing of network messages making use of the open source simulation framework OMNeT++ [4], which is free for non-profit use and highly modular. The OMNeT++ framework comes with a built-in graphical user interface (GUI) that displays network topologies, nodes and messages providing some debug functions that allow a deeper inspection of message contents and node variables.

- Underlying network model

  This simulation framework was designed in order to support different kinds of underlying network models. For that reason OverSim has three implemented network models:

  - The Simple network model sends data packets directly from one overlay node to another by using a global routing table, being the most scalable model.
  - The SimpleHost model was implemented in order to reuse overlay protocol implementations without code modifications in real networks. In this model each OverSim instance emulates only a single host. These OverSim instances can be connected to other instances over existing networks like the Internet.
  - The INET underlay model that is derived from the INET framework of OMNeT++, which includes simulation models of all network layers from the MAC

layer to the higher layers. This network model allows the simulation of complete backbone structures.

- Overlay protocols

  The framework comes with several overlay protocols already implemented. Most of them are structured peer-to-peer protocols but unstructured peer-to-peer protocols are available as well. In order to facilitate the implementation of new overlay protocols this framework provides several functions that many overlay protocol implementations have in common. The communication between overlay and application makes use of the Common API for Structured Overlays [26].Overlay protocols that want to use this API have to provide at least a key-based routing interface (KBR) to the application and the overlay protocols that use a distributed hash table (DHT) can offer this service to the application using the same interface.

- Applications

  The use of the Common API design allows the evaluation of a wide range of different applications that rely on key-based routing with exchangeable overlays.

OverSim has been able to run large-scale simulations of overlay networks with up to 100000 nodes with two different underlying network models (Simple and INET).

Chapter 3

ITS Overlay Framework

3.1 Overview

The **ITS Overlay Framework Architecture** is basically constituted by three main elements:

- **Overlay**

  The Overlay element is where the nodes connect themselves in different groups sending messages to each other in their own group. This block contains the ITS overlay nodes.

- **Group Server**

  In the Group Server all the possible groups are defined. The Groups Server's function is to assign groups and provide group information to nodes when they want to enter an overlay and when they move (change their position) while connected to the overlay.

- **Simulator**

  The Simulator provides the necessary data to the nodes. The position of the nodes in the overlay is sent to them by the Simulator.

These **Framework Elements** are further discussed and explained in the next section.

The **ITS Overlay Framework Elements** (**Overlay**, **Group Server** and **Simulator**) are divided in three fundamental layers:

  – Network Communications Layer

  – Management Layer

  – Application-Level Layer

The **Network Communications Layer** provides the methods to exchange messages between the nodes in the same group and also between the Overlay, the Group Server and the Simulator (see Figures 3.1 and 3.2). As it will be further explained in the **Network Communications Layer**, these three blocks exchange requests between themselves using HTTP protocol thus defining in this layer the URL contexts to which each block responds.

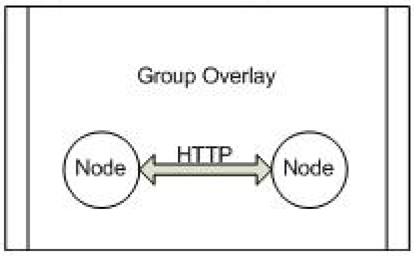

Figure 3.1: Network Communication Between Nodes

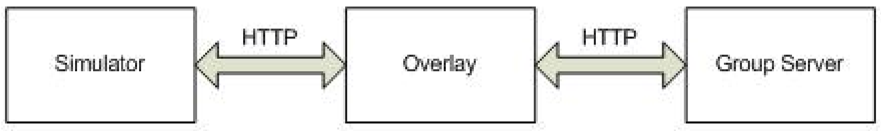

Figure 3.2: Framework Elements

The **Management Layer** provides the methods for establishing an overlay with a determined network model thus determining which requests (using the defined URL contexts in the **Network Communications Layer**) and messages are exchanged between nodes in a group overlay and between the Overlay block and the Group Server and Simulator blocks.

The **Application-Level Layer** provides a means to create different group based overlays without modifying the network model of the overlay thus providing the methods to test an overlay network model in different scenarios. All the new classes that are created to implement a group are encapsulated inside a **GroupsCreator** object.

This framework is an object oriented framework implemented in JAVA programming language that provides its classes and interfaces in a layered scheme as we can see in Figure 3.3. For the development of the framework we used the Eclipse Integrated Development Environment (IDE). Eclipse [39] is an open-source software framework written primarily in Java. It is an IDE for Java developers, but users can extend its capabilities by installing plug-ins written for the Eclipse software framework, such as development toolkits for other programming languages like Python and C/C++.

For the development of the framework's architecture we consider that the underlying network for the overlay network will mainly use the 3G communication technology to

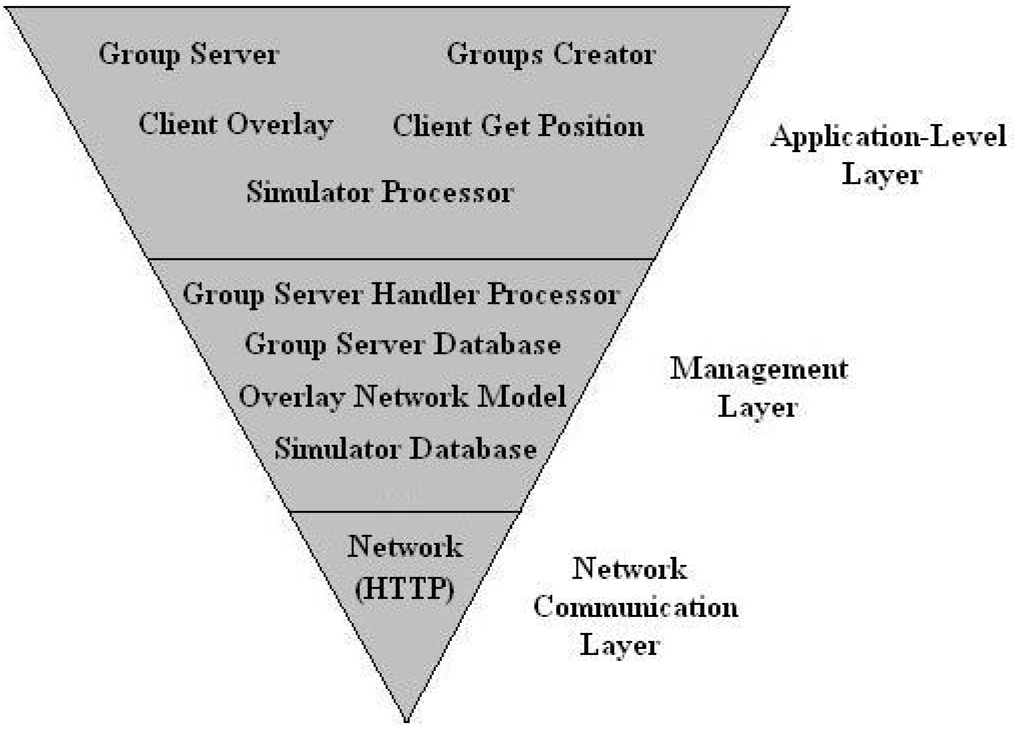

Figure 3.3: Framework Layers

establish connections. We created the network model for the overlay networks based on this assumption.

3.2 Framework Elements

As said in the **Overview** the **ITS Overlay Framework** has three distinct blocks which will be described in this section:

- Overlay
- Group Server
- Simulator

3.2.1 Overlay

The Overlay is the block of the Framework Architecture where the nodes and the overlay network model are defined.

In truly decentralized overlay networks (e.g. peer-to-peer networks such as Gnutella and Freenet), there is no central server. All the nodes on the overlay network act as both a server and a client. We defined interfaces for implementing simple networking models based on this paradigm with the idea that each node can act as both a server and a client. Therefore a user can develop his own overlay network model using the provided interfaces.

3.2.1.1 Overlay Nodes

A node is characterized by its unique identification number (**NodeID**) and its position, speed and direction. The NodeID is an unique number assigned by the Group Server when a node joins a group. The other data can be either get by a microscopic traffic simulator or by a GPS.

According to this we have created structure class for the nodes:

- Node ID - An unique serial number to identify the node in the overlay;
- Position in X - An integer with the position of the node in the X axis;
- Position in Y - An integer with the position of the node in the Y axis;
- Speed - An integer with the speed of the node;
- Direction - An integer with the direction of the node;
- Central - An auxiliary boolean variable;
- Node IP - A string with the IP address of the node;

In the implemented examples of overlay networks only the position was used. The speed and direction implementations are for future work.

So if we want to create a **Node** object we must specify its NodeID and its position. The **Node** class also provides two significant methods:

- void updateNode(int NodeID, int x, int y, String IP) - This method allows a Node object to have its values updated.
- String toURI() - Method to return a String with the node information to use in an HTTP request. The returned String has the following format: "NodeID=int&X=int &Y=int&IP=String".

When a node begins its execution it starts an HTTP server that responds to HTTP requests from other nodes. This way a node acts as both a server and a client since all nodes can send requests to other nodes. A detailed explanation of this operation is presented in the **Network Communications Layer** section.

3.2.1.2 Operation of an Overlay Node

1. Getting the Position

   The position is essential for the nodes to enter the group overlays. To provide the nodes with their position the interface **GetPosition** was developed. This interface fetches pair of position values X and Y to provide them separately to the nodes. Using this interface the nodes can get their position from various sources like a traffic simulator or a GPS.

   The most important methods of this interface are:

   - void fetchXY()
   - int getX()
   - int getY()

   In order to test the framework we developed the **GetPositionFromSimulator** class that implements this interface. This class fetches the position values from the implemented Simulator using HTTP requests. This interface and class will be further discussed on the **Application-Level Layer** section.

2. Joining the Overlay

   When a node begins executing the first step to enter the overlay is to get their position. Their position can be get from the **GetPosition** interface either by a traffic simulator or by other means like GPS. In the development of the framework we only used position data given from a simulator. Being given the position, the node sends a request to enter a group to the **Group Server** containing the position in order for the server to identify in which group the node enters. Then the **Group Server** assigns an unique ID to the node and returns information about the group assigned to the node.

   When the node knows its group it informs all the other nodes in the same group overlay that it has joined the group sending a join message that reaches all the nodes in the group.

3. Sending Messages in the Overlay

   While being in the group overlay the node is able to send messages to the nodes in the same group overlay. The node can send messages to an individual node or send broadcast messages that reach the entire group overlay network.

4. Node's Update in the Overlay

   While connected to the overlay the node keeps on asking for its position using the **GetPosition** interface. Since the node's position keeps on changing the node must update its data in the **Group Server** to guarantee it still is in the same group and must also update the group overlay to inform all other nodes of its new position. Therefore the when the node gets a new position it sends an update node request to the **Group Server** and receives information of its actual group if it has not changed or receives information of its new group otherwise.

If the node stays in the same group it sends an update message to all the other nodes in the group. If a new group is assigned to the node by the **Group Server**, the node sends a message to all nodes in the old group informing that it is leaving the group. Then it has to send messages all nodes in the new group to inform that it is joining the group.

5. Leaving the Overlay

   When a node wants to leave the overlay it has to request the **Group Server** to delete the node information in the server database and then has to send messages to all nodes in the group informing that it is leaving the group.

3.2.1.3 Overlay Network Model

The ITS Overlay Framework allows the use of different network topologies by implementing two interfaces in the node client application: the **OverlayProcessor** and the **HandlerProcessor**.

These interfaces are based on a network model that uses five distinct functions with four types of requests, to the five different situations defined before in the operation of the overlay nodes (see Figure 3.4):

- Join a group (/join);
- Update the node in the group (/update);
- Send a message to an individual node in the group (/sendMsg);
- Send broadcast messages (/sendMsg with destination NodeID as -1);
- Leave the group (/leave);

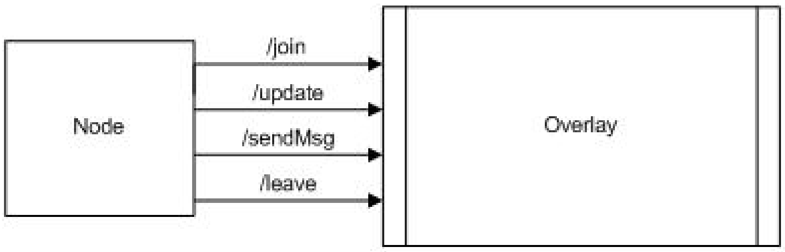

Figure 3.4: Overlay Network Model

The **OverlayProcessor** and the **HandlerProcessor** interfaces methods and functions are explained in detail in the **Management Layer** section.

3.2.2 Group Server

The main function of the Group Server is to create the groups with which the nodes will be assigned and respond to the nodes requests.

There are four basic types of requests from the nodes to the Group Server (see Figure 3.5):

- Add Node(/addnode)

  Request from a node to enter an overlay. The node sends its position and is assigned with an unique ID and a group.

- Update Node (/updatenode)

  Request from a node to update its position in the server. Based on the nodes position the server assigns the same group or a new group to the node.

- Operation (/operation)

  Request from a node that asks the server for an operation defined by the user.

- Remove Node (/delnode)

  Request from a node when it leaves the overlay network to remove its information from the server.

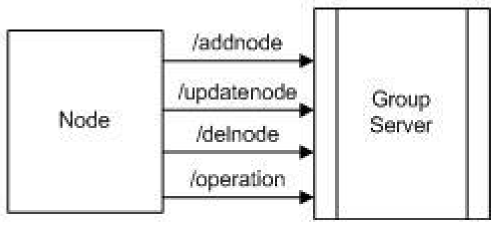

Figure 3.5: Requests from a Node to the Group Server

The Group Server starts an HTTP server in order to process the HTTP requests from the nodes and respond to them. The processing of the requests and format of the responses are made through the **ServerHandlerProcessor** interface on the **Group Server**. This interface is discussed further in the **Management Layer** section.

Groups

Each group has its own identification number (GroupID) and is defined by its borders. Using the same position scale of X and Y as the nodes each group has a maximum and minimum value of X and Y. Therefore all the basic groups have rectangular borders. Although different types of groups such as groups that covers crossings and curves can be defined by assigning the same GroupID to multiple adjacent basic rectangular groups.

Therefore a group in the **Group Server** has the following structure class:

- Group ID - A serial number to identify the group;
- GroupNodes - Vector that contains the nodes that are in the group;
- CentralNode - A node in the group chosen by the server that can act as a central node in an overlay;
- Borders - Contains the maximum and minimum values X and Y for that group.

To create a **Group** object we must then specify its GroupID and its borders (xmin, xmax, ymin and ymax variables). A group is assigned to a node when the node's X and Y positions are equal or higher to the minimum X and Y values of the group borders and lower than the maximum X and Y values of the group borders. The nodes added to the "GroupNodes" vector are of the **Node** object described in the **Overlay Nodes** paragraph.

The most important methods the **Group** class provides are:

- void addNode(Node node) - Adds the provided node to the group. If the central boolean of the node is true then it is also the central node.
- void updateNode(int NodeID, Node newnode) - Method to update the node with the provided NodeID with the parameters provided in the newnode object.
- String toString() - Builds a string containing the group and its nodes information.
- Node getNodeByID(int NodeID) - Method to get the node in the group with the provided NodeID.
- Node getLastUpdatedNode() - Method to get the last updated node.
- int getMinNodeID() - Method to get the minimum NodeID value in the group.
- int getMaxNodeID() - Method to get the maximum NodeID value in the group.

In order to create different types of groups the interface **GroupsCreator** was implemented. This interface is discussed on the **Application-Level Layer** section.

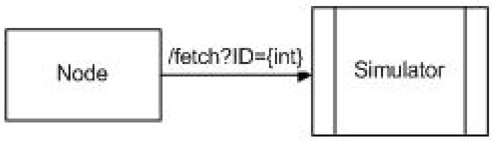

Figure 3.6: Request from a Node to the Simulator

3.2.3 Simulator

The function of the Simulator is to provide position for the nodes. The Simulator runs an HTTP server that responds to requests from the nodes. The request is a simple fetch for the position of the nodes with a determined ID (different from NodeID).

The Simulator responds to the "/fetch" requests giving the position of the node with the required ID (see Figure 3.6). The response format was designed in order to interact with the **GetPositionFromSimulator** class that was implemented in the **Overlay Node**. This interaction is discussed in the **Management Layer** and **Application-Level Layer** sections.

3.3 Framework Architecture Layer Scheme

3.3.1 Network Communications Layer

The Network Communications Layer is the lowest layer in the framework architecture. This layer provides the nodes with the means to connect themselves to other nodes in an overlay.

3.3.1.1 Overlay

The communication protocol used to connect the nodes in overlays is the HTTP protocol. A node acts as both HTTP client and HTTP server.

Node HTTP Client

The HTTP requests in a lower level use a class licensed to the Apache Software Foundation (ASF), the **ElementalHttpPost** class. Using this class we implemented a "post(URI)" method that allows the nodes to post HTTP requests.

In order to facilitate the usage of this class we implemented the **Communication** class on a higher level using the **ElementalHttpPost**. This class provides a simple means for the nodes to send requests to the Group Server and to other nodes.

Therefore the main methods of this class are:

- String sendToServer(String URI)
- String sendToNode(Node node, String URI)

Node - Group Server Connection

The "sendToServer(String URI)" method allows the nodes to send its requests to the **Group Server**. There are four basic types of requests from the nodes to the **Group Server**:

- /addnode
- /updatenode
- /operation
- /delnode

Since the requests are already defined, we implemented in the top level of the **Network Communication Layer** the **GroupServerRequests** class. This class uses the **Communication Class** to provide static methods to directly send requests to the **Group Server**:

- String AddNode(int x, int y)
- String UpdateNode(int x, int y)
- String LeaveNode()
- String Operation()

The three methods are called every time a node joins, updates and leaves a group overlay during the client overlay execution. The "Operation()" method is an optional method for nodes to get auxiliary information from the **Group Server**.

Node - Node Connection

The "sendToNode(Node node, String URI)" method allows the nodes to send its requests to other nodes. There are four basic types of requests from the nodes to other nodes:

- /join
- /update
- /sendMsg
- /leave

Like in the "Node - Group Server Connection" case we implemented in the top level of the **Network Communication Layer** the **OverlayRequests** class to provide nodes a simple means to send these requests to other nodes. This class also uses the **Communication Class**. The methods of this class are all static as they can be called by other classes to send requests to a node:

- String join(Node DestNode, Node JoinNode, String Headers)
- String update(Node DestNode, Node UpdateNode, String Headers)
- String leave(Node DestNode, Node LeaveNode, String Headers)
- String sendMsg(Node DestNode, int SenderID, int DestID, String message, String Headers)
- String transmitNode(Node node, Node destnode, String type, String Headers)

These methods were developed in order to be called by implementations of the **OverlayProcessor** and **HandlerProcessor** interfaces.

Node - Simulator Connection

The connection between a node and the **Simulator** does not use the **Communication** class but instead it is defined in the **GetPositionFromSimulator** class that implements the **GetPosition** interface.

When a node runs the method "fetchXY()" of the class a "/fetch" request is sent to the **Simulator** using the post method from the **ElementalHttpPost** class.

In Figure 3.7 we have the three levels for the **Network Communications Layer**.

Node HTTP Server

Each node starts an HTTP server when it enters the overlay. It is a method to received HTTP requests from other nodes. Since each node has an unique ID, the NodeID, each node starts its HTTP server listening on PORT where PORT equals to 8000 plus the NodeID of the node. This way, when testing overlays on one computer each node is listening on a different unique port.

To start an HTTP server on a node, we must specify an implementation of the **HandlerProcessor** interface. As already described in the **Framework Elements** section, this interface is in charge of process the requests from other nodes.

The HTTP server is implemented in order handle the four types of URL contexts presented before (see Figure 3.8):

- /join

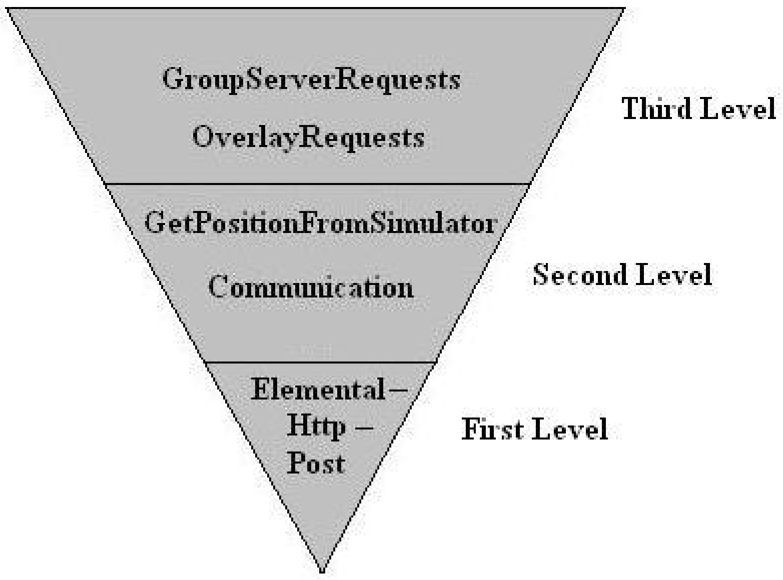

Figure 3.7: Node Client Network Communications Architecture

- /update
- /sendMsg
- /leave

All of these requests are processed by the provided implementation of the **Handler-Processor** interface.

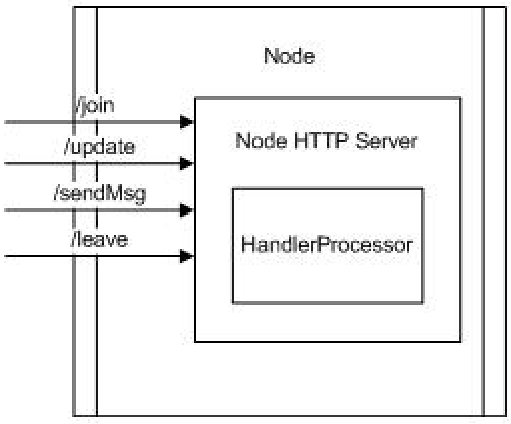

Figure 3.8: Node Acting as Server

3.3.1.2 Group Server

The **Group Server** is in charge of creating the groups for the overlays and assigning these groups to nodes based on the nodes position. Therefore, on the communications point of view, the **Group Server** works only as a server that responds to the HTTP requests from the nodes.

The **Group Server** starts its HTTP server the same way a node does. To start the HTTP server of the **Group Server** we must specify an implementation of the **Server-HandlerProcessor** interface and the port where we want the server to listen. As already described in the **Framework Elements** section, this interface is in charge of process the requests from nodes and provide them with their group information.

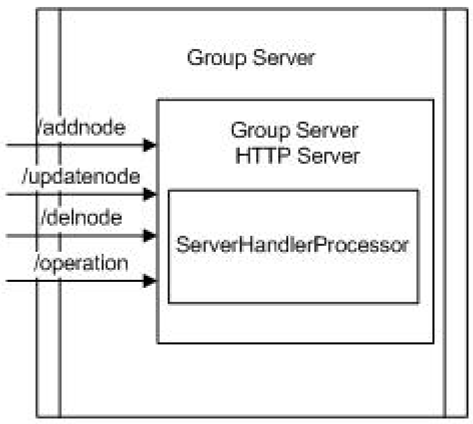

Figure 3.9: Group Server HTTP Server

So in the **Group Server** the HTTP server is implemented in order handle the four types of URL contexts presented before in the **Node HTTP Client** (see Figure 3.9):

- /addnode
- /updatenode
- /operation
- /delnode

3.3.1.3 Simulator

The **Simulator** also works as an HTTP server in order to provide the position when a node requests it. The Simulator HTTP server only process requests on the defined URL Context: /fetch (see Figure 3.10). For the **Simulator** to begin its HTTP server we must specify an implementation of the **SimulatorDatabase** interface and the port where we want the server to listen.

3.3.2 Management Layer

The **Management Layer** takes care of the management of nodes in the **Overlay**,of the **Group Server** and defines the how they respond to requests and also takes care of how the **Simulator** gets its simulation data.

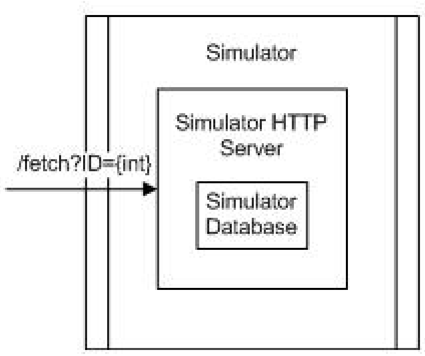

Figure 3.10: Simulator HTTP Server

So this layer tasks include routing and addressing of nodes in their group overlays and process of the group server to the nodes requests. As already referred on the **Framework Elements** section, the interfaces in charge of these tasks are the **ServerHandlerProcessor**, the **OverlayProcessor**, the **HandlerProcessor** and the **SimulatorDatabase**.

3.3.2.1 Group Server Management

The interface in charge of processing the requests from the nodes and deliver the desired responses to them is the **ServerHandlerProcessor** interface. We have implemented a class using this interface called **DefaultServerHandlerProcessor** in order to exemplify how to use the interface. This class is presented and explained on the **Default Implementations** section.

The groups defined in the **Group Server** are saved on a static Vector variable of groups, the "Vector¡Group¿ Groups", in the **GroupServerDatabase** class. This class provides the methods with which the implementations of the **ServerHandlerProcessor** interface process the incoming requests from the nodes.

We will now describe with more detail the **ServerHandlerProcessor** interface and the **GroupServerDatabase** class.

ServerHandlerProcessor

According to the types of requests defined in the **Network Communications Layer** section the interface main methods for this interface are:

- String addNode(String request, String IP)

  Method to process the "/addnode" request from a node by giving it a NodeID and assigning it to a group and then finally return a response the node. The response should contain node and group information in order for the node to enter a group overlay.

- String updateNode(String request, String IP)

  Method to process the "/updatenode" request from a node verifying if the node has leaved a group and entered other and then return a response to it. This method returns an identical response to the "addNode" method response.

- String delNode(String request)

  This method processes the "/delnode" request and is in charge of removing the node form the server database and then returns a response informing if the node was successfully deleted.

- String operation(String request)

  This is a method designed to process a user defined operation responding to the "/operation" request. It returns the desired response to the request.

GroupServerDatabase

As referred before, this class contains the groups defined by the user through an implementation class of the **GroupsCreator** interface and each group contains its nodes. So this class provides the methods to change the groups and change its nodes:

- Group getGroupByID(int GroupID)

  Method that returns the Group with the provided GroupID.

- Group getGroupByNodeID(int NodeID)

  Method that returns the Group object that contains the node with the provided NodeID.

- int getGroupIndByXY(int x, int y)

  Method that returns the index of the group in the vector that contains the provided coordinates.

- void setGroupByID(int GroupID, Group sub)

  Method to replace the group saved in the vector with the provided GroupID with the provided "sub" group.

- int addNode(Node node)

  Method that adds a node to the respective group and returns the GroupID of the assigned group.

- int updateNode(Node node)

  Method that updates the node with the NodeID of the provided node with the provided node object and returns the GroupID of the updated node's group.

- void delNode(int NodeID, int GroupID)

  Method that removes the node with the provided NodeID from the group with the provided GroupID.

3.3.2.2 Overlay Nodes Management

Overlay Network Model - Interfaces

As referred before on the **Framework Elements** section, the methods to process the responses from the group server and that define the network topology to use in the group overlays are defined by the **OverlayProcessor** and **HandlerProcessor** interfaces.

OverlayProcessor

The **OverlayProcessor** interface is in charge of establishing communication in the overlay with the other nodes in the group.

This interface's first function is to process the responses from the Group Server to the add node and update node requests. Therefore the implementation classes of this interface need to have the updated position from the nodes to provide it in the requests from the nodes.

The second function of the **OverlayProcessor** interface is to define the requests that the nodes send between themselves (using the already defined URL contexts in **Network Communications Layer**) thus allowing the use of different network topologies together with the HandlerProcessor interface.

The most important methods of this interface are related to the five types of requests the nodes can send:

- int processResponseAddNode(String response)

  Method to process a response from the **Group Server** to an add node request. While processing the response the node joins the indicated group overlay.

- int processResponseUpdateNode(String response)

  Method to process a response from the **Group Server** to an update node request. The processing of the response includes updating the node in the group overlay or leaving the actual group and joining a new group overlay depending on the **Group Server** response.

- String groupLeaveNode()

  Method to inform the group overlay that the node is leaving the group.

- String sendMsg(String message, int NodeID)

  Method to send a message to a node with the given NodeID in the group overlay.

- String sendBroadcast(String message)

  Method to send a broadcast message to all nodes in the same group overlay.

HandlerProcessor

The **HandlerProcessor** interface determines the response from the various nodes to the requests from other nodes. By creating classes that implement this interface we can have different topologies and routing schemes.

The methods from this interface are destined to interact with the requests defined in the overlay network model. Therefore the main methods of this interface are:

- String joinGroup(String input, String IP)

  Method to process a join group request from a node that is joining the group overlay.

- String updateGroup(String input, String IP)

  Method to process a update node request from a node that is updating its position in the group overlay.

- String leaveGroup(String input)

  Method to process a leave group request from a node that is leaving the group.

- String messageHandler(String request)

  Method to process the received messages from a node. These messages can be either broadcast messages or direct messages.

3.3.2.3 Simulator Management

Getting the Simulation Data

In order to get the simulation data (in this case only the position) an interface was developed in the simulator that contains the methods the simulator uses to get the position from a text file or from an implementation of an already existing simulator. This interface is named **SimulatorDatabase** and its main methods are:

- void fetchXY() - Fetches a new pair of values X and Y for each node in the simulation;
- int getX(int ID) - Returns the value X of a simulated node;
- int getY(int ID) - Returns the value Y of a simulated node;
- String buildResponse(String input) - Builds a string with position values of a node to return via an HTTP response.

Getting the Simulation Data from a Text File

The developed simulator has already an implementation class of the **Simulator-Database** interface that fetches the simulation data from a text file although it only gets the position since no function to get the speed and direction of the nodes was implemented. This class is called **SimulatorDatabaseTextFile**.

The format of the text file read by the simulator with four nodes is:

```
ID=1&X=1&Y=1  ID=2&X=1&Y=3    ID=3&X=1&Y=6    ID=4&X=2&Y=3
ID=1&X=1&Y=2  ID=2&X=1&Y=4    ID=3&X=1&Y=7    ID=4&X=2&Y=2
ID=1&X=1&Y=3  ID=2&X=1&Y=5    ID=3&X=1&Y=8    ID=4&X=2&Y=1
```

For each node to be simulated an "ID=int&X=int&Y=int" must be added to the file. Each line contains a single position for each node. Each position from one node must be separated by a tab in each line.

When the "fetchXY()" method is called, the **SimulatorDatabaseTextFile** class reads the next line from the text file obtaining the position of the nodes contained in the text file.

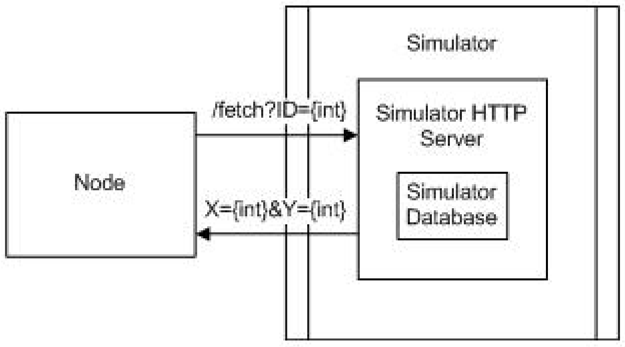

Figure 3.11: Node-Simulator Connection

The **Simulator** responds to the nodes requests with their X and Y position values that were last fetched from the text file using the "buildResponse(String input)" method which returns the string "X=int&Y=int" for the **SimulatorDatabaseTextFile** class as we can see in Figure 3.11. As referred before, the response has this format in order to interact with the **GetPositionFromSimulator** class from an **Overlay Node**.

3.3.3 Application-Level Layer

The **Application-Level Layer** is the higher layer of the framework's architecture layered design. This layer provides the required methods through classes and interfaces in order to run the **Overlay** nodes applications, to run the **Group Server** constructing the desired groups and to run the **Simulator** defining where it gets its simulation data.

3.3.3.1 Overlay Nodes

ClientOverlay

We developed the **ClientOverlay** class so that we can execute an overlay node application. This class provides methods to initialize a node application thus allowing them to enter a group overlay (see Figure 3.12). To instantiate a **ClientOverlay** object the host and port of the **Group Server** are needed as well as an implementation of the **OverlayProcessor** interface. The group information and other nodes information as well is saved in this class in a static ”Group” object.

The main methods of this class are:

- int addNode(int x, int y)

  Method to send a request to add the node with its position X and Y to the **Group Server** and then process its response using the provided implementation class of the **OverlayProcessor** interface.

- int updateNode(int x, int y)

  Method to send a request to update the node with its updated position X and Y to the **Group Server** and then process its response using the provided implementation class of the **OverlayProcessor** interface.

- String leaveNode()

  Method to inform the group overlay that the node is leaving using the provided implementation class of the **OverlayProcessor** interface and then send a request to the Group Server to remove the node.

- String sendMsg(String message, int NodeID)

  Method to send a message to a node in the same group overlay with the destination NodeID.

- String sendBroadcast(String message)

  Method to send a broadcast message to all the nodes in the same group overlay.

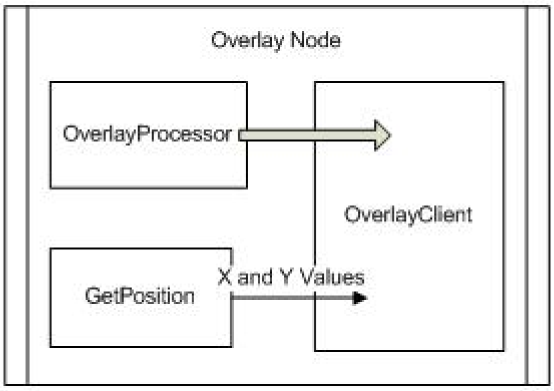

Figure 3.12: Overlay Node Application-Level

GetPosition

The **ClientOverlay** needs the nodes position in order to allow the node to enter a group overlay. We can get the position from a simulator or a GPS using the **GetPosition** interface. There is already an implemented class of this interface to get the position from the **Simulator**, the **GetPositionFromSimulator** class. The methods of this class are:

- void fetchXY()

  Fetches the next pair of values X and Y by sending a "/fetch" request to the **Simulator**. The **Simulator** responds with the next pair of position values X and Y of the node.

- int getX()

  Returns the last fetched value of X for the node.

- int getY()

  Returns the last fetched value of Y for the node.

3.3.3.2 Group Server

In order to run a **Group Server** application we implemented the **GroupServer** class. But before the server starts its operation as an HTTP Server to respond to the nodes requests it must create the groups using the **GroupsCreator** interface.

GroupsCreator

The interface **GroupsCreator** was implemented in order to create different types of groups . The main method for this interface is the "Vector¡Group¿ getGroups()" method.

The implementation classes of this interface must implement this method in order to return a vector of group objects described in the **Framework Elements** section containing the groups that are assigned by the **Group Server** to the nodes.

GroupServer

After getting an implementation class of the **GroupsCreator** interface that defines the desired groups, the **GroupServer** class can be instantiated. To instantiate this class we must not only use the **GroupsCreator** class but also an implementation of the **ServerHandlerProcessor** interface responsible of processing the requests from the nodes and also specify the port to which the Group Server HTTP server will listen (see Figure 3.13).

In this class we have a static object of the **GroupServerDatabase** class to whom it is provided the implementation class of the **GroupsCreator** interface.

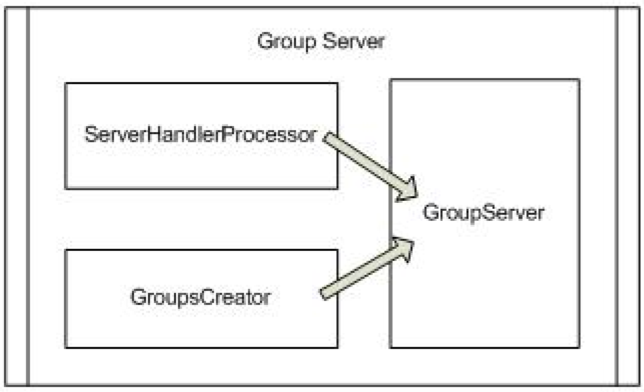

Figure 3.13: Group Server Application-Level

The **GroupServer** class has an implemented method to change the implementation class of the **GroupsCreator** interface it uses in order to define different groups:

- String setGroups(GroupsCreator gc) - Creates a new **GroupServerDatabase** with the provided **GroupsCreator** object;

3.3.3.3 Simulator

A class was implemented to allow the execution of the **Simulator** application called **SimulatorProcessor**. To instantiate this class an implementation class of the **SimulatorDatabase** interface like the **SimulatorDatabaseTextFile** class and integer with the port number where the Simulator HTTP request will listen are needed.

The **SimulatorProcessor** class provides the method "fetchXY()"to get new position values with which the **Simulator** provides the nodes when they request for their position.

The new position values are obtained from the implementation class of the **SimulatorDatabase** interface. In the case of the **SimulatorDatabaseTextFile** class, each time the "fetchXY()" method is called, a new line is read from the text file containing the new position values for the nodes.

Chapter 4

Framework Test Scenarios

4.1 Default Implementations

To perform the required tests to the framework and to give an example of the implementation of the necessary interfaces to write and run **ITS Overlay Applications** we have developed default classes that implement the interfaces.

Some of these default classes were already presented and explained in the **Framework Architecture Layer Scheme** sections. The already presented default classes are the **GetPositionFromSimulator** class that implements the **GetPosition** interface in the **Overlay Nodes** and the **SimulatorDatabaseTextFile** class that implements the **SimulatorDatabase** interface in the **Simulator**.

The other developed default classes implement the following interfaces: the **ServerHandlerProcessor** and **GroupsCreator** in the **Group Server** and the **OverlayProcessor** and **HandlerProcessor** in the **Overlay Nodes**.

Some of these interfaces interact with each other, so their implementation classes also interact with each other. The implementation class of the **ServerHandlerProcessor** interface provides the responses to the node's requests and the implementation class of the **OverlayProcessor** interface processes those responses and it also defines the requests the node's send between themselves which are process by the implementation class of the **HandlerProcessor** interface. So the implementation classes for these three interfaces must be developed together.

4.1.1 Group Server

For the **Group Server** we have one default implementation for each of its two interfaces: the GroupsCreator interface and the **ServerHandlerProcessor** interface. These default implementations also work as examples for users to create their own implementation classes.

4.1.1.1 Default GroupsCreator Implementations

The default implemented class for this interface is named **RoadGroups**. To call this class we have to specify the number of horizontal groups H, the number of vertical group Y, and the size in X and Y for the groups. So this class creates the desired number of adjacent groups of equal size in the way described in Figure 4.1:

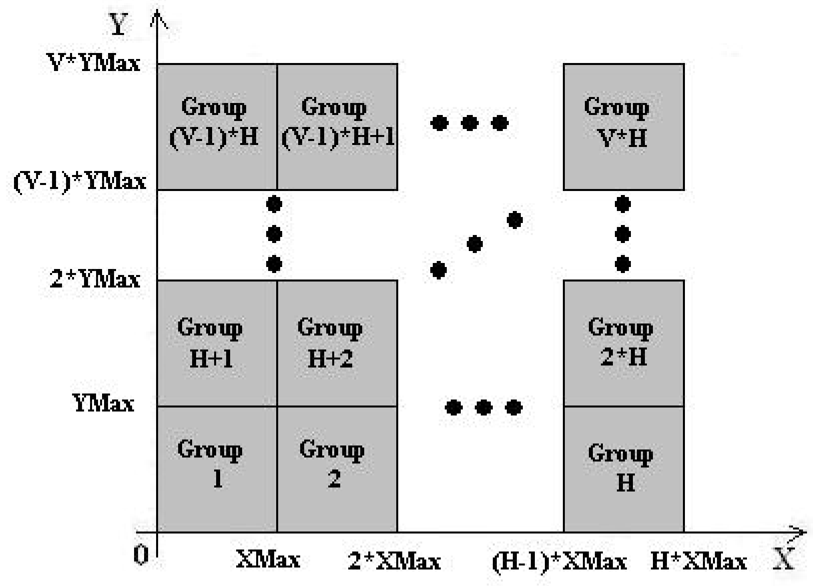

Figure 4.1: Road Groups

4.1.1.2 Default ServerHandlerProcessor Implementation

We developed the **DefaultServerHandlerProcessor** class that interacts with the default implemented class of the **OverlayProcessor** interface of the **Overlay Node**. The **DefaultServerHandlerProcessor** will be described using the **ServerHandlerProcessor** interface explanation in the **Management Layer** section as a background.

Processing the Requests

- Add Node

  The implemented class is prepared to receive add node requests with the format "/addnode?X=int&Y=int". The class creates a new node with the position given in the request and assigns a unique NodeID to the node. Then, based on the position of the node it assigns a group to the node.

Since this class was implemented in order to interact with the implemented **DefaultOverlayProcessor** class of the **OverlayNode**, it is also responsible for select a **Central Node**. This function is already implemented in the **GroupServerDatabase** class in the "addnode(Node node)" method so that if the group does not have any node it assumes that the node that is entering the group is the **Central Node**.

Finally, a response based on the node and its group is returned. The response has the following format:

```
ThisNode
NodeID=1
Group
GroupID=1
XMin=0
XMax=5
YMin=0
YMax=5
CentralNode
NodeID=1
X=1
Y=1
IP=127.0.0.1
Nodes
NodeID X Y IP
1 1 1 127.0.0.1
```

If the node is not successfully added to the group server database it returns a "-ERR" response.

- Update Node

  The format of update node requests that this class is prepared to process is /updatenode?NodeID=int&X=int&Y=int where the NodeID refers to the unique ID of the node and the given position is the new position of the node. The class modifies the position of the node with the given NodeID and based on the new position of the node it verifies if the node stays in the same group or if it is necessary to assign a new group to the node. The "lastupdate" timestamp variable of the node is also updated.

  The responses format is the same as the response to the add node request.

- Delete Node

  In order for this class to remove a node from the group server database it requires its NodeID. So the delete node requests have the format "/delnode?NodeID=int

&GroupID=int" giving the NodeID of the node to remove and the additional information of the GroupID of the node, allowing to reduce the processing time in finding the node.

If the node is successfully deleted from the group server database the response is "+OK NodeDeleted" else it returns a "-ERR" response.

- Operation

  Since this class was implemented in order to interact with the implemented **DefaultOverlayProcessor** class of the **OverlayNode** interface. The "Operation" function is to get a new **Central Node** for the desired group. Only the **Central Node** in a group makes this request when it desires to leave the group. The request format is "/operation?GroupID=int". Then a new **Central Node** is selected based on the "last update" variable of the nodes in the group. The response format is "NodeID=int" where the given value corresponds to the NodeID of the new **Central Node**.

4.1.2 Default Overlay Network Topology

We developed a default overlay network topology using the **OverlayProcessor** and **HandlerProcessor** interfaces described in the **Management Layer** section.

The **DefaultOverlayProcessor** and **DefaultHandlerProcessor** implemented classes interact with the **DefaultServerHandlerProcessor** class that implements the **ServerHandlerProcessor** interface from the **Group Server**.

The node and its group informations are all put in a static group object named **Group** in the **ClientOverlay** class. This way the **Group** object is accessible to all the classes.

The default implemented classes use common methods to process the responses from the **Group Server** to the nodes requests. For that reason we developed a class of static methods that can be of help to create new overlay network topologies, the **OverlayUtils** class.

Overlay Network Topology

The default network topology is a centralized topology. Every node that enters and is in the group overlay sends messages to a **Central Node** and then the **Central Node** takes care of routing and broadcast (see Figure 4.2).

- Joining a Group

  Whenever a node enters the group overlay, if the group does not have any nodes, the node itself becomes the **Central Node**. From that point every node that joins the group is a **Normal Node**,so it sends a join request to the **Central Node** that

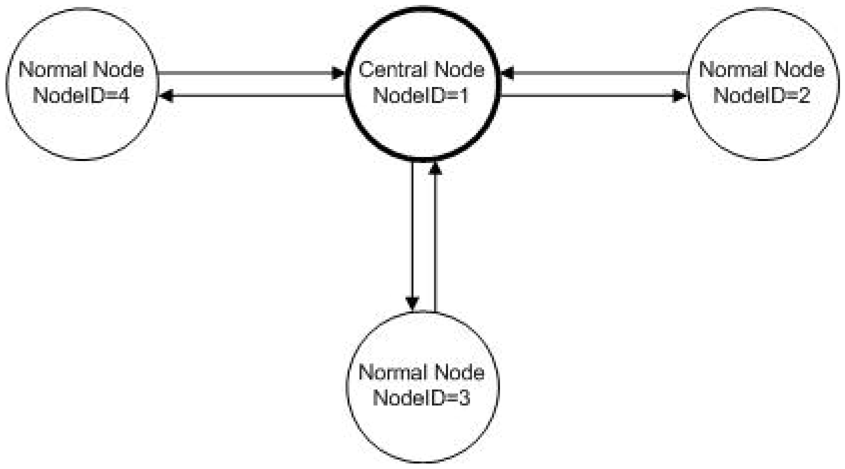

Figure 4.2: Centralized Topology with Four Nodes

propagates the join request to all the other nodes informing every node in the group that a new node joined the group.

- Sending Messages

  A node can send messages in two ways: an individual message to one node in the group and a broadcast message to all nodes in the group using -1 as the destination NodeID value. If the node is the **Central Node** of the group then it sends the messages directly to the node with the desired NodeID or sends broadcast messages directly to all the nodes in the group overlay. If the node is a **Normal Node** then it sends the messages to the **Central Node** who is in charge of forward the messages to the desired node if it is an individual message, or to all the nodes if it is a broadcast message.

- Leaving a Group

  When a node wants to leave a group overlay we have two possible cases. Case the node is a normal node it sends a leave group request to the **Central Node** and then the request is sent by the **Central Node** to all the other nodes in the group overlay and the node successfully leaves. Case the node is the **Central Node** it asks the **Group Server** for a new **Central Node** using the "/operation" request and then inform all the nodes in the group overlay that it is leaving the group and which node will be the new **Central Node**.

- Updating Node in Group

  When a node updates its position in the **Group Server** it can stay in the same group overlay or get assigned with a new group. If the node stays in the group and is a **Normal Node** it sends an update node in group request to the **Central Node** and then the **Central Node** sends the request to all the other nodes, updating the position of the node that sent the update request in every node of the group overlay. If the node is the **Central Node** it simply sends an update request to all the other nodes in the group overlay. If the node is assigned with a new group it leaves the group as described before in "Leaving a Group" and joins a new group also as described before in "Joining a Group".

4.1.2.1 Overlay Management Utilities

All methods of this class are static and were created in order to help the development of classes that implement the **OverlayProcessor** and **HandlerProcessor** interfaces considering that the **Group Server** uses the implemented **DefaultServerHandlerProcessor** class.

The most relevant methods of this class for the **OverlayProcessor** classes are:

- Node getThisNode(String response, int x, int y)

  Gets the node's NodeID using the response from the **Group Server** and then returns a node object created using it and the provided node's position.

- Node getCentralNode(String response)

  Gets the Central Node data using the response from the **Group Server** and then returns a node object with the retrieved data.

- int getGroupID(String response)

  Gets the node's GroupID using the response from the **Group Server** and returns it.

- void addGroupNodes(String response)

  Retrieves the data from all the nodes of the group overlay creating node objects with it and then adds each node object to the static **Group** object in the **ClientOverlay** class.

- void NodeHTTPServer(HandlerProcessor pc)

  Starts the Node HTTP Server to process overlay requests from other nodes according to the provided implementation class of the **HandlerProcessor** interface.

- void defineGroup(int GroupID)

  Empties the static **Group** object of the **ClientOverlay** class and sets its new GroupID.

- String getThisNodeIP()

  Returns a String with the node's IP.

The other methods of this class were implemented in order to interact with the **HandlerProcessor** classes:

- Node getNode(String input, String IP, boolean aux)

  Returns a node object with the data get from that node request (the IP can be get via HTTP connection if the aux boolean is true). The node requests formats are defined on the **OverlayProcessor** classes.

- Node getNode(String input)

  Gets the NodeID from node that sent the request and gets that node in the **GroupNodes** vector of the static **Group** object returning it.

- int getData(String input, String type)

  Method that parses a node's query and gets the integer value where the parameter equals the type String (e.g if type equals "NodeID", when the node receives a "/join?NodeID=1&X=1&Y=1" this method returns an integer with value equal to 1).

- String getDataString(String input, String type)

  This method does the same as the one before but it returns a String instead of an integer (e.g. if type equals "NodeID", when the node receives a "/join?NodeID=1&X=1 &Y=1" this method returns a String equal to "1").

4.1.2.2 DefaultOverlayProcessor Class

When a node begins operating the first thing it does is to send an add node request to the **Group Server**. The implemented **DefaultServerHandlerProcessor** class responds with a string containing the node and group information (assigned NodeID, assigned group and its nodes information)and identifies the central node of the group (the format of the response is on the **Group Server Management** explanation).

- Process the response from the **Group Server** to an add node request

  Having the response, the "processResponseAddNode(String response)" method of the **DefaultOverlayProcessor** class first gets the node's assigned GroupID from the response using the static "getGroupID(String response)" method from the **OverlayUtils** class and if the GroupID value is allowed it uses the static "defineGroup(int GroupID)" method from the **OverlayUtils** class.

  Then it uses the static "getThisNode(String response, int x, int y)" method that returns a node object of the node. This node object is put on the **ThisNode** variable of the static **Group** object.

  Next the method gets the node object of the **Central Node** using the static "getCentralNode(String response)" method and puts it in the **CentralNode** variable of the static **Group** object.

Finally the method runs the static "addGroupNodes(String response)" method of the **OverlayUtils** class in order to put all the nodes information as nodes objects in the vector **GroupNodes** of the static **Group** object.

The node has now all the information of its group overlay available so it starts its operation as an HTTP server using a static method from the **OverlayUtil** class, the "NodeHTTPServer(HandlerProcessor pc)" method using the **DefaultHandler-Processor** class to define its HTTP server responses to the requests.

If the node is the **Central Node** of the group this method does nothing more, else if the node is not the Central Node of the group it sends a join group request to the **Central Node** with the following format: "/join?NodeID=int&X=int&Y=int &IP=String" where the values in the request are those of the node.

- Leave the Group Overlay

  When a node wants to leave its group overlay if it is a **Normal Node** it sends a leave group request to the **Central Node** with the following format: "/leave?NodeID=int".

  If the node is the **Central Node** of its group it first sends an operation request to the **GroupNode** asking to assign a new **Central Node** and using the response it gets the new **Central Node** NodeID. Then it informs all the nodes in the group that it is leaving indicating the new **Central Node** NodeID in the leave group request. So, the leave group request sent by the node has the following format: "/leave?NodeID=int&NewCentralNodeID=int".

- Process the response from the **Group Server** to an update node request

  Each time the node gets its position values updated it can inform the **Group Server** sending an update node request to it. The **Group Server** responds the same way it responds to the add node requests.

  The method gets the new GroupID from the response and if it is the same as the actual GroupID, the node has to inform all other nodes in the group overlay of the update. So, if the node is a **Normal Node** it sends an update node in group request to the **Central Node**, else if the node is the **Central Node** of the group it sends and update node in group request directly to all the other nodes in its group overlay.

  If the new GroupID is different from the actual GroupID of the node, the node uses the "groupLeaveNode()" method and then does the same proceedings as in the "processResponseAddNode(String response)" method joining the new group overlay.

- Send Message

  When a node wants to send a message to another node in the group it runs this method. This method codes the message to send so that all of the characters in the message are allowed in the HTTP request. Next, if the node is the Central Node it sends the message directly to the node with the given NodeID, else if the node is a **Normal Node**.

The format of the send message request is "/sendMsg?FROM=int & TO=int & MSG=String" where in the "FROM" is put the NodeID of the node that sends the message, in the "TO" is put the NodeID of the destination node and in the "MSG" camp is put a String with the coded message.

- Send Broadcast Message

  This method allows the nodes in a group to send broadcast messages to all the other nodes in the group. Like the send message method this method codes the messages to send so that all of the characters in the message are allowed in the HTTP request. The format of the broadcast message request is the same as in the normal message request.

  If the node that wants to send the broadcast message is a **Normal Node** then it sends a message with the "TO" camp set as -1 to the **Central Node**. If the node is the **Central Node** it uses the "sendMsg(String message, int NodeID)" to send the message to all the other nodes in the group.

4.1.2.3 DefaultHandlerProcessor Class

- Process a join group request

  When a node receives a join group request from another node it runs this method. First the method runs the static "getNode(String input, String IP, boolean aux)" method that returns a node object of the node that sent the request. Then it adds the node object to **GroupNodes** vector of the static **Group** object.

  If the node that received the request is the **Central Node** of the group, it sends the join request to all the other nodes in the group overlay thus informing all nodes that a new node has joined the group overlay.

- Process an update node in group request

  When a node receives an update group request from another node this method is run. This method is very similar to the previous one as it first runs the static "getNode(String input, String IP, boolean aux)" method that returns a node object of the node that sent the request. Then it replaces the old node object in the **GroupNodes** vector of the static **Group** object with the new node object.

  If the node that received the request is the **Central Node** of the group, it sends the update request to all the other nodes in the group overlay thus informing all nodes of the update of a node in the group overlay.

- Process a leave group request

When a leave group request is received by a node, this method gets the NodeID from the node that is leaving using the static "getData(String input, type)" method from the **OverlayUtils** class with type equal to "NodeID".

Then, if the retrieved NodeID is not equal to the **Central Node** NodeID the node in the **GroupNodes** vector with the retrieved NodeID is removed. Else, if the retrieved NodeID is equal to the Central Node NodeID, the method gets the new **Central Node** NodeID using again the static "getData(String input, type)" method but with type equal to "NewCentralNodeID" and then sets the new **Central Node** and remove the old one from the static **Group** object.

- Message handler

  When a node receives a send message request this method gets the sender node NodeID and the destination node NodeID from the request using the static "getData(String input, type)" method with type equals to "FROM" and "TO" respectively and also gets the message using the static "getDataString(String input, type)" method with type equals to "MSG".

  The method then verifies if the node that received the request is the destination node comparing the NodeIDs and if it is, the message is decoded and thus delivered.

  If the node is not the destination node and it is a **Normal Node** it does nothing more. But if it is a **Central Node**, it sends the message to the node with the previously gotten destination NodeID or if the destination NodeID is -1 it sends the message to all the nodes in the group except the sender node.

4.2 How to Write A New Application

The **ITS Overlay Framework** provides the methods to create Group Overlays where nodes connect themselves based on their position. This section describes how to use the framework to develop **ITS Overlay Applications**. In order to run a **Overlay Node Client Application** we also need to run a **Group Server Application** and if we use simulation data to supply the nodes with their position, then we also have to run a **Simulator Application**.

4.2.1 ITS Overlay Applications

The method for creating an **ITS Overlay Application** is the same for all the applications:

- First we need to define the how the overlay will work.
- Afterwards we need to implement classes for the necessary interfaces in each application according to the previous step.

- Finally we create the application by using the correspondent classes to begin each application described in the **Application-Level Layer** section in the **ITS Overlay Framework** chapter and using the implemented classes in the previous steps.

4.2.2 Group Server Application

In order to create a **Group Server Application** we first need to define the groups to create with the **GroupsCreator** interface and also define the responses of the **Group Server** to the overlay node's requests with the **ServerHandlerProcessor** interface. After doing these steps we can create a **Group Server Application** using the **GroupServer** class presented in the **Application-Level Layer** section and the created classes.

Example

First we have to implement the class **GroupsCreatorExample** using the **GroupsCreator** interface. In the method "Vector¡Group¿ getGroups()" provided in the interface we must create a vector object and add the desired groups to the vector and then return the vector of groups:

```
public class GroupsCreatorExample implements GroupsCreator{
        public Vector<Group> getGroups() {
                Vector<Group> Groups = new Vector<Group> ();
                Groups.add(new Group(GroupID, Xmin, Xmax, Ymin, Ymax));
                Groups.add(new Group(GroupID, Xmin, Xmax, Ymin, Ymax));
                return Groups;
        }
}
```

Secondly we use the class **DefaultServerHandlerProcessor** that implements the **ServerHandlerProcessor** interface (see **Default Implementations** section).

Now that we have an implementation class of the **GroupsCreator** and **ServerHandlerProcessor** interface, we want our server to listen to the port 12000, then we write a **Group Server Application** the following way:

```
public class GroupServerApplicationExample {
        public static void main(String[] args) {
             GroupsCreator gc = new GroupsCreatorExample();
             ServerHandlerProcessor pc = new ServerHandlerProcessorExample();
             new GroupServer(gc,pc, 12000);
        }
}
```

Now that we have written the application we just have to run it using a Java Virtual Machine (JVM) and we have the **Group Server** ready to process the node's requests and assign one of the created groups to them.

4.2.3 Overlay Node Client Application

To create an **Overlay Node Client Application** the first thing to do is to define the network topology by creating classes that implement the **OverlayProcessor** and **HandlerProcessor** interfaces. Afterwards we need to get the position for the nodes. The position can be get either by manual input or using the **GetPosition** interface.

After doing these steps we can write an **Overlay Node Client Application** using the **ClientOverlay** class presented in the **Application-Level Layer** section and using the created classes.

Example

First we have to define the network topology and write the classes that implement the **OverlayProcessor** and **HandlerProcessor** interfaces. For this example we will use the default implementations of these interfaces named **DefaultOverlayProcessor** and **DefaultHandlerProcessor**.

In this example we assume we get the nodes positions through the **Simulator** using the default implementation of the **GetPosition** interface, the **GetPositionFromSimulator** class.

Assuming a **GetPositionFromSimulator** is running in the "192.168.1.253" IP address listening to the port 12000 and the **Simulator** is running in the "192.168.1.252" IP address listening to the port 14000, we can now write the **Overlay Node Client Application** the following way:

```
public class OverlayNodeClientApplicationExample {
        public static void main(String[] args) {
                OverlayProcessor pc = new DefaultOverlayProcessor();
                ClientOverlay client = new ClientOverlay("192.168.1.253",
                    12000, pc);
                GetPosition pos = new GetPositionFromSimulator("
                    192.168.1.252", 14000, ID);
                pos.fetchXY();
                client.addNode(pos.getX(), pos.getY());
                while(pos.getX()!=-1 && pos.getY()!=-1 ){
                        pos.fetchXY();
                        client.updateNode(pos.getX(), pos.getY());
                        client.sendBroadcast("message");
                }
                client.leaveNode();
        }
}
```

Now that we have written the application we just have to run it for each node changing the ID that they use to get their position from the server.

4.2.4 Simulator Application

As already described in the **Application-Level Layer** section, to run a **Simulator Application** we must use the **SimulatorProcessor** class. This class needs a implementation class of the **SimulatorDatabase** interface in order to get the simulation data, so the first step in the creation of a **Simulator Application** is to define how the simulator gets its data (from text file, from traffic flow algorithms, etc.).

Afterwards we have to create a **SimulatorDatabase** class so that the simulator gets the simulation data in the defined way. Finally we need to define the simulator behaviour (like the interval of time it updates its data or the number of times in which the simulator provides nodes with their position) in order to write the application.

Example

In this example we consider that the node's get its position from the default implementation of the **SimulatorDatabase** interface, the **SimulatorDatabaseFromTextFile** class. This class is better explained in the **Management Layer** section.

Supposing we have the simulation data in the "nodes.txt" file, that the simulator will accept request on port 14000 and that we want the simulator to update the position of the nodes each five seconds we write a **Simulator Application** for this example in the following way:

```
public class SimulatorApplicationExample {
    public static void main(String args[]) throws InterruptedException {
        SimulatorDatabase db = new SimulatorDatabaseTextFile("nodes.txt")
            ;
        SimulatorProcessor pc = new SimulatorProcessor(db, 14000);
        while(!pc.isEnd()){
                    pc.fetchXY();
                    Thread.sleep(5000);
        }
        pc.setEnd();
    }
}
```

4.3 Testing Methodology

The methodology to test scenarios can be divided in five steps:

- Define the scenario;
- Prepare the scenario conditions;
- Write the necessary applications;

- Run the written applications;
- Evaluate the test scenario;

To better understand this methodology the proceedings of the steps referred above will now be described separately.

- Define the scenario:
  - Number and types of groups;
  - Number of nodes;
  - Messages exchanged between nodes;
  - Overlay network topology;
- Prepare the scenario conditions:
  - How the nodes get their position;
  - Implement a **GroupsCreator** interface class to create the required groups;
  - Define the **Group Server** operation using the **ServerHandlerProcessor** interface.
  - Define the overlay network topology using the **OverlayProcessor** and **HandlerProcessor** interfaces.
- Write the necessary applications:
  - Write the **Group Server Application**;
  - Write the **Overlay Node Client Application** that the nodes will run;
  - Write a **Simulator Application** if necessary;
- Run the written applications:
  - Run the developed **Group Server Application** verifying if it properly creates the desired groups;
  - Run the **Simulator Application** if it is needed;
  - Run an **Overlay Node Client Application** for each node;
- Evaluate the test scenario:
  - Verify if the defined scenario has successfully worked;

4.4 First Test Scenario

4.4.1 Test Specifications

According to the methodology to test scenarios the first step is to define the scenario. Then we prepare the scenario conditions.

4.4.1.1 Scenario Definition

For this first scenario we only need to create one group. The group will have a size of 2 x 20 in the used variables (X and Y) as we can see in Figure 4.3.

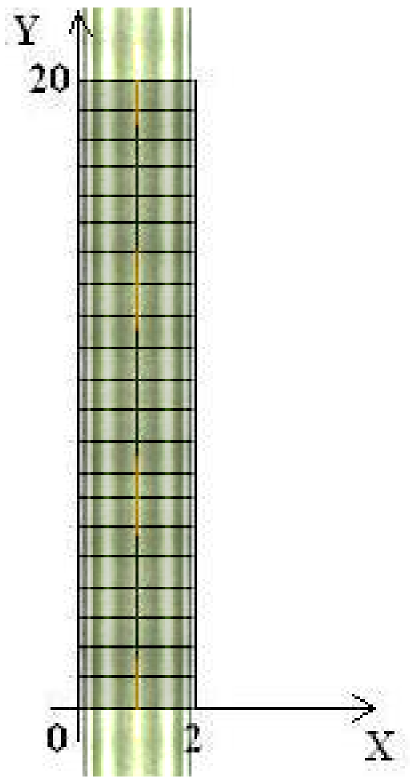

Figure 4.3: Road Group

We will use five vehicles as nodes. One of the nodes will have an accident and sends a broadcast message to the group overlay.

For the group overlay network topology we will use the default topology of the framework, a centralized topology.

4.4.1.2 Scenario Conditions

The overlay nodes should get their position through a **Simulator** using the **GetPositionFromSimulator** class each five seconds.

The simulation data will be on a text file named "first_test_scenario.txt" so the **Simulator** will use the **SimulatorDatabaseTextFile** class to get the simulation data. In the file we have the position for the five nodes in fifteen intervals of time of the defined five seconds. The simulation data that is used in this scenario is on Table 4.1.

In the **Group Server** we will also use the default implementation class **RoadGroups** for creating the defined group and use the **DefaultServerHandlerProcessor** class to define the responses of the **Group Server**.

ID=1	**ID=2**	**ID=3**	**ID=4**	**ID=5**
X=1 Y=1	X=1 Y=3	X=1 Y=6	X=2 Y=11	X=2 Y=16
X=1 Y=2	X=1 Y=4	X=1 Y=7	X=2 Y=10	X=2 Y=15
X=1 Y=3	X=1 Y=5	X=1 Y=8	X=2 Y=9	X=2 Y=14
X=1 Y=4	X=1 Y=6	X=1 Y=9	X=2 Y=8	X=2 Y=13
X=1 Y=5	X=1 Y=7	X=1 Y=10	X=2 Y=7	X=2 Y=12
X=1 Y=6	X=1 Y=8	X=1 Y=11	X=2 Y=6	X=2 Y=11
X=1 Y=7	X=1 Y=9	X=1 Y=12	X=2 Y=5	X=2 Y=10
X=1 Y=8	X=1 Y=10	X=1 Y=13	X=2 Y=4	X=2 Y=9
X=1 Y=9	X=1 Y=11	X=1 Y=14	X=2 Y=3	X=2 Y=8
X=1 Y=10	X=1 Y=12	X=1 Y=15	X=2 Y=2	X=2 Y=7
X=1 Y=11	X=1 Y=13	X=1 Y=16	X=2 Y=1	X=2 Y=6
X=1 Y=12	X=1 Y=13	X=1 Y=17	X=2 Y=0	X=2 Y=5
X=1 Y=13	X=1 Y=13	X=1 Y=18	X=2 Y=-1	X=2 Y=4
X=1 Y=14	X=1 Y=13	X=1 Y=19	X=2 Y=-1	X=2 Y=3
X=1 Y=15	X=1 Y=13	X=1 Y=-1	X=2 Y=-1	X=2 Y=2

Table 4.1: First Test Scenario Simulation Data

Since we use the default centralized topology provided by the framework, when writing the **Overlay Node Client Application** we will use the **DefaultOverlayProcessor** and **DefaultHandlerProcessor** classes.

When we write the **Overlay Node Client Application** we will also define that the node that has the accident is the node with ID equal to two.

Now that we have the required implementation classes we can write the applications.

4.4.2 Writting the Applications

According to the defined scenario and implemented interfaces we now write the applications for the **Group Server**, the **Overlay Nodes** and the **Simulator**. We will do the tests in the same machine so the **Group Server** and the **Simulator** host is on "localhost". The **Group Server** will listen to port 12000 and **Simulator** will listen to port 14000.

4.4.2.1 Group Server Application

Since we use the **RoadGroups** and **DefaultServerHandlerProcessor** classes in the **Group Server** for this scenario. To call a **RoadGroups** object we must specify the number of horizontal groups and vertical groups as one and the vertical size as twenty one and horizontal size as three. So, this **Group Server Application** is written in the following way:

```
public class FirstScenarioGroupServerApplication {
        public static void main(String[] args) throws IOException {
            GroupsCreator gc = new RoadGroups(1, 1, 3, 21);
            ServerHandlerProcessor hpc = new DefaultServerHandlerProcessor();
            GroupServer Server = new GroupServer(gc, hpc, 12000);
        }
}
```

4.4.2.2 Simulator Application

As already defined we use the **SimulatorDatabaseTextFile** to get the simulation data from the "first_test_scenario.txt" text file and we want our **Simulator** to update the positions of the nodes each five seconds, so we write the **Simulator Application** in the following way:

```
public class FirstScenarioSimulatorApplication {

   public static void main(String args[]) throws InterruptedException {
            SimulatorDatabase db = new SimulatorDatabaseTextFile("
                first_test_scenario.txt");
            SimulatorProcessor pc = new SimulatorProcessor(db, 14000);
            while(!pc.isEnd()){
               pc.fetchXY();
               Thread.sleep(5000);
            }
            pc.setEnd();
   }
}
```

4.4.2.3 Overlay Nodes Client Application

We already have the implemented classes necessary to run an **Overlay Node Client Application**, but we also need to define the behaviour of each node. Instead of running a different type of application for each node, since they use an ID value to get their position from the **Simulator** we can run the same application but with different entry values. So we will write the application so that we need to input the IDs of the nodes to begin its execution. Also, using an "if" clause we can define which node transmits the accident message and when it transmits the message.

The node we have considered to have the accident is the node with the ID 2. So according to provided simulation data we can see that the node position stays the same after the tenth iteration.

We can now write the following **Overlay Node Client Application**:

```
public class FirstScenarioOverlayNodeClientApplication {

        public static void main(String[] args) throws InterruptedException {
            if(args.length<1){
```

```
            System.out.println("Put ID");
        }
        System.out.println(">> Starting Overlay Node Client");
        int ID = Integer.parseInt(args[0]);
        int i=1;
        OverlayProcessor pc = new DefaultOverlayProcessor();
        ClientOverlay client = new ClientOverlay("localhost", 12000, pc);
        GetPosition pos = new GetPositionFromSimulator("localhost",
            14000, ID);
        pos.fetchXY();
        client.addNode(pos.getX(), pos.getY());
        while(pos.getX()!=-1 && pos.getY()!=-1 ){
            Thread.sleep(5000);
            i++;
            pos.fetchXY();
            client.updateNode(pos.getX(), pos.getY());
            if(ID==2 && i>10)
                client.sendBroadcast("AccidentInX"+pos.getX()+"Y"+pos.
                    getY());
        }
        client.leaveNode();
    }
}
```

4.4.3 Results and Conclusions

Having written the **Overlay Node Client Application**, the **Group Server Application** and the **Simulator Application** we have now the conditions to test if the defined scenario works. We now create a jar file for each of these applications: "nodeclient.jar", "groupserver.jar" and "simulator.jar".

We begin to run the **Group Server Application** and the **Simulator Application** using the "java -jar" command plus the file name of the correspondent application and verify if these applications run as expected.

Then we execute an **Overlay Node Client Application** for each node in the simulation indicating their position through the ID input.

4.4.3.1 Running the Group Server

Having started the **Group Server Application** we verify that the defined group for this scenario was successfully created in the application output:

```
> java -jar groupserver.jar

Group
GroupID=1
XMin=0
XMax=2
```

```
YMin=0
YMax=20

>> Starting HTTP Server
```

4.4.3.2 Running the Simulator

When we run the **Simulator Application** we can verify through its output that the server is processing a different position for each node in five second intervals in a total of fifteen iterations. Below we have the output of the simulator for the first five iterations:

```
> java -jar simulator.jar

>> Starting HTTP Server
>> Processing line:
ID=1&X=1&Y=1 | ID=2&X=1&Y=3 | ID=3&X=1&Y=6 | ID=4&X=2&Y=11 | ID=5&X=2&Y=16
================
>> Processing line:
ID=1&X=1&Y=2 | ID=2&X=1&Y=4 | ID=3&X=1&Y=7 | ID=4&X=2&Y=10 | ID=5&X=2&Y=15
================
>> Processing line:
ID=1&X=1&Y=3 | ID=2&X=1&Y=5 | ID=3&X=1&Y=8 | ID=4&X=2&Y=9 | ID=5&X=2&Y=14
================
>> Processing line:
ID=1&X=1&Y=4 | ID=2&X=1&Y=6 | ID=3&X=1&Y=9 | ID=4&X=2&Y=8 | ID=5&X=2&Y=13
================
>> Processing line:
ID=1&X=1&Y=5 | ID=2&X=1&Y=7 | ID=3&X=1&Y=10 | ID=4&X=2&Y=7 | ID=5&X=2&Y=12
================
```

4.4.3.3 Running the Overlay Nodes

To execute an **Overlay Node Client Application** in a node we must specify the ID. For example if we want a node to have the positions defined in the simulation data by the ID 1 we run the following command: " $java - jaroverlaynode.jar1$"

So we execute an application for each of the five nodes in this scenario each with a different ID from 1 to 5.

We verify that all the nodes enter the network and that the first node that enters the group is the **Central Node** of the group. We can see this information in the nodes output displaying the **Group Server** responses:

```
Group
GroupID=1
XMin=0
XMax=3
YMin=0
YMax=21
```

```
CentralNode
NodeID=1
X=1
Y=1
IP=127.0.0.1
Nodes
NodeID X Y IP
1 1 1 127.0.0.1
2 2 3 127.0.0.1
3 1 6 127.0.0.1
4 2 11 127.0.0.1
5 2 16 127.0.0.1

==================
```

When the simulation reaches the tenth iteration the node with the ID 2 sends a broadcast message to the group through the **Central Node** (node with NodeID 1 in this simulation).

```
===============
>> Request Host: http://127.0.0.1:8001
>> Request URI: /sendMsg?FROM=2&TO=-1&MSG=AccidentInX1Y13
<< Response: HTTP/1.1 200 OK
===============
```

The **Central Node** processes the message and sends it to all the other nodes in the group. We can see its output:

```
===============
>>Processing Message...
>> Request Host: http://127.0.0.1:8003
>> Request URI: /sendMsg?FROM=2&TO=3&MSG=AccidentInX1Y13
<< Response: HTTP/1.1 200 OK
===============
>> Request Host: http://127.0.0.1:8004
>> Request URI: /sendMsg?FROM=2&TO=4&MSG=AccidentInX1Y13
<< Response: HTTP/1.1 200 OK
===============
>> Request Host: http://127.0.0.1:8005
>> Request URI: /sendMsg?FROM=2&TO=5&MSG=AccidentInX1Y13
<< Response: HTTP/1.1 200 OK
===============
>> Received Broadcast Message: AccidentInX1Y13
+OK BroadcastDelivered (-0)
```

4.4.3.4 Conclusions

The implementation of this test scenario using the **ITS Overlay Framework** has successfully worked because the **Simulator** provided the simulation data from the text file to the correspondent nodes and the defined group was created and assigned to the nodes by the **Group Server**.

Also the nodes entered the correct group overlay and did all their position updates in the group overlay and the node that had the accident has sent the broadcast message to the **Central Node** and the **Central Node** sent the message to all the other nodes in the group overlay, so the centralized topology was implemented with success.

4.5 Second Test Scenario

4.5.1 Test Specifications

4.5.1.1 Scenario Definition

The objective for this second test scenario is to test a different kind of groups. We will create three different groups that represent a cross road with semaphores. These groups are represented in Figure 4.5:

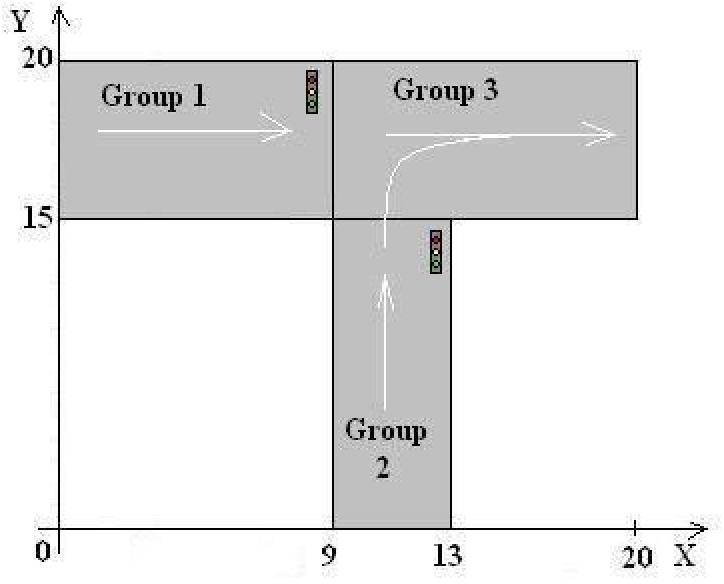

Figure 4.4: Crossing Groups

We will use six vehicles as nodes. In the beginning of the test we will have two nodes stopped in the group 1 semaphores sending messages to the group indicating a semaphore is red and a third node entering the group. The group 3 does not have any node and the group 2 has one node at the semaphore that immediately passes to group 3, one node going in the middle of the group and one node entering the group. These two final nodes will stop at the semaphore of the group 2 because it turns red. Figure 4.5 displays the initial positions of the nodes in their groups.

Initially the nodes 1 and 2 are stopped at the semaphore and the node 3 is going in their direction and will also stop at the semaphore. The node 4 has green light to go on

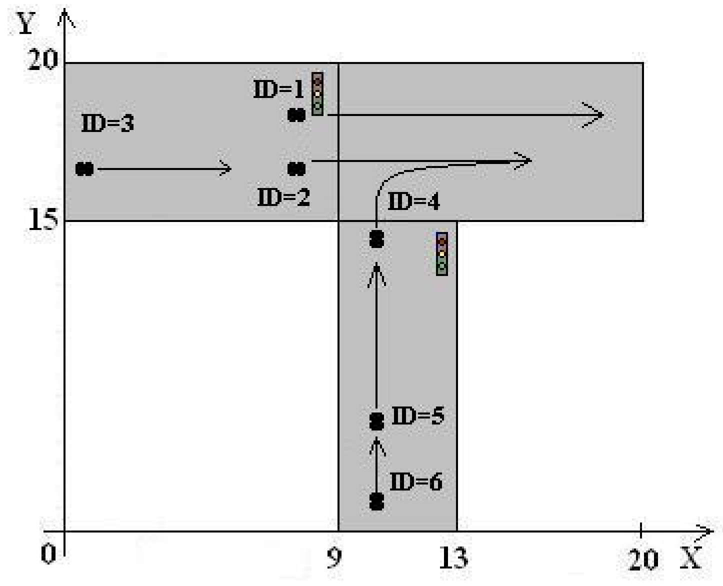

Figure 4.5: Nodes' Initial Position in the Groups

and goes begins moving to group 3. The nodes 5 and 6 are moving straight in group 2 and then the semaphore turns red and they eventually stop at the semaphore. When the semaphore in group 2 turns red, the semaphore in group 1 turns green and then the nodes 1, 2 and 3 will move forward to group 3. Finally when the semaphore in group 1 turns red, the semaphore in group 2 turns green and the nodes 5 and 6 proceed to group 3.

4.5.1.2 Scenario Conditions

This scenario conditions are similar to those of the previous scenario. The overlay nodes get their position from the **Simulator**. The simulation data is in a text file named "second_test_scenario.txt" with fifteen positions for each of the six nodes. The simulator updates its values each five seconds during the seventeen time intervals. The positions for the nodes in the text file are in Table 4.2.

In the **Group Server** we use the **DefaultServerHandlerProcessor** class to define the responses. We will also use the same overlay network topology as in the previous test scenario, the default centralized topology. So we will use the **DefaultOverlayProcessor** and **DefaultHandlerProcessor** classes when writing the **Overlay Node Client Application**. We just need to implement the **GroupsCreator** interface of the **Group Server** in order to create the desired groups. We create the SecondTestScenarioGroups class that implements the **GroupsCreator** interface and in the "getGroups()" method of the class we add the three group objects with the defined coordinates to a vector and then return that vector:

```
public class SecondTestScenarioGroups implements GroupsCreator{

        public Vector<Group> getGroups() {
```

ID=1	**ID=2**	**ID=3**	**ID=4**	**ID=5**	**ID=6**
X=8 Y=16	X=8 Y=18	X=1 Y=16	X=11 Y=14	X=11 Y=5	X=11 Y=1
X=8 Y=16	X=8 Y=18	X=2 Y=16	X=11 Y=15	X=11 Y=6	X=11 Y=2
X=8 Y=16	X=8 Y=18	X=3 Y=16	X=12 Y=16	X=11 Y=7	X=11 Y=3
X=8 Y=16	X=8 Y=18	X=4 Y=16	X=13 Y=16	X=11 Y=8	X=11 Y=4
X=8 Y=16	X=8 Y=18	X=5 Y=16	X=14 Y=16	X=11 Y=9	X=11 Y=5
X=8 Y=16	X=8 Y=18	X=6 Y=16	X=15 Y=16	X=11 Y=10	X=11 Y=6
X=8 Y=16	X=8 Y=18	X=7 Y=16	X=16 Y=16	X=11 Y=11	X=11 Y=7
X=8 Y=16	X=8 Y=18	X=7 Y=16	X=17 Y=16	X=11 Y=12	X=11 Y=8
X=8 Y=16	X=8 Y=18	X=7 Y=16	X=18 Y=16	X=11 Y=14	X=11 Y=9
X=9 Y=16	X=9 Y=18	X=8 Y=16	X=19 Y=16	X=11 Y=14	X=11 Y=10
X=10 Y=16	X=10 Y=18	X=9 Y=16	X=-1 Y=-1	X=11 Y=14	X=11 Y=11
X=11 Y=16	X=11 Y=18	X=10 Y=16	X=-1 Y=-1	X=11 Y=14	X=11 Y=12
X=12 Y=16	X=12 Y=18	X=11 Y=16	X=-1 Y=-1	X=11 Y=14	X=11 Y=13
X=13 Y=16	X=13 Y=18	X=12 Y=16	X=-1 Y=-1	X=11 Y=15	X=11 Y=14
X=14 Y=16	X=14 Y=18	X=13 Y=16	X=-1 Y=-1	X=12 Y=16	X=11 Y=15
X=15 Y=16	X=15 Y=18	X=14 Y=16	X=-1 Y=-1	X=13 Y=16	X=12 Y=16
X=16 Y=16	X=16 Y=18	X=15 Y=16	X=-1 Y=-1	X=14 Y=16	X=13 Y=16

Table 4.2: Second Test Scenario Simulation Data

```
                Vector<Group> Groups = new Vector<Group>();
                Groups.add(new Group(1, 0, 9, 15, 20));
                Groups.add(new Group(2, 9, 13, 0, 15));
                Groups.add(new Group(3, 9, 20, 15, 20));
                return Groups;
        }
}
```

4.5.2 Writting the Applications

According to the defined scenario and the implemented interfaces we now write the applications for the **Group Server**, the **Overlay Nodes** and the **Simulator**. We will do the tests in the same machine so the **Group Server** and the **Simulator** host is on "localhost". The **Group Server** will listen to port 12000 and **Simulator** will listen to port 14000.

4.5.2.1 Group Server Application

For this scenario we use the created **SecondTestScenarioGroups** class that provides the created groups and we will be using the **DefaultServerHandlerProcessor** class to define the **Group Server** responses. So, the **Group Server Application** for this test scenario is written in the same way as the in the first test scenario but using the **SecondTestScenarioGroups** as the **GroupsCreator** object.

4.5.2.2 Simulator Application

We write the **Simulator Application** the same way as in the first scenario. We just have to modify the text file where the **Simulator** gets the simulation data to the "second_test_scenario.txt" text file. So we create a new **SimulatorDatabase** object with the **SimulatorDatabaseTextFile** class using the "second_test_scenario.txt" as an input.

4.5.2.3 Overlay Nodes Client Application

We write the **Overlay Node Client Application** in the same way as in the first test scenario using the ID as an input, but for this particular scenario instead of a node that has an accident, we consider that the nodes that stop at the semaphores send a message to the other nodes in the group. So we will use an "if clause" to make the nodes send a message when their position is the same in two consecutive iterations. Considering this, the **Overlay Node Client Application** for this scenario can be written in the following way:

```
public class SecondScenarioOverlayNodeClientApplication {

        public static void main(String[] args) throws InterruptedException {
        if(args.length<1){
                System.out.println("Put ID");
        }
        System.out.println(">> Starting Overlay Node Client");
        int ID = Integer.parseInt(args[0]);
        OverlayProcessor pc = new DefaultOverlayProcessor();
        ClientOverlay client = new ClientOverlay("localhost", 12000, pc);
                GetPosition pos = new GetPositionFromSimulator("localhost",
                    14000, ID);
                pos.fetchXY();
                int X = pos.getX();
                int Y = pos.getY();
                client.addNode(pos.getX(), pos.getY());
                while(pos.getX()!=-1 && pos.getY()!=-1 ){
                        Thread.sleep(5000);
                        pos.fetchXY();
                        client.updateNode(pos.getX(), pos.getY());
                        if(pos.getX()==X && pos.getY()==Y)
                                client.sendBroadcast("SemaphoreIsRed!");
                        X = pos.getX();
                        Y = pos.getY();
                }
                client.leaveNode();
        }
}
```

4.5.3 Results and Conclusions

Using the written applications for this particular scenario we can test if the defined scenario works. Like for the first test scenario we now create a jar file for each of the created applications: "nodeclient.jar", "groupserver.jar" and "simulator.jar".

We begin to run the **Group Server Application** and the **Simulator Application** using the "java -jar" command. Then we execute an **Overlay Node Client Application** for each of the six nodes in this scenario indicating their simulation data through the ID input.

4.5.3.1 Running the Group Server

When we begin the **Group Server Application** execution we verify the groups in the application output:

```
> java -jar groupserver.jar

Group
GroupID=1
XMin=0
XMax=8
YMin=15
YMax=20

Group
GroupID=2
XMin=9
XMax=13
YMin=0
YMax=14

Group
GroupID=3
XMin=9
XMax=20
YMin=15
YMax=20

>> Starting HTTP Server
```

The three defined groups were successfully created by the implemented **SecondTestScenarioGroups** class. The **Group Server** is now ready to process and respond to the nodes requests.

4.5.3.2 Running the Simulator

When we run the **Simulator Application** we can verify through its output that the server is processing a different position for each node in five second intervals in a total of seventeen iterations. Below we have the output of the simulator for the first five iterations:

```
> java -jar simulator.jar

>> Processing line:
ID=1&X=8&Y=16 | ID=2&X=8&Y=18 | ID=3&X=6&Y=16 | ID=4&X=15&Y=16 | ID=5&X=11&Y
    =10  | ID=6&X=11&Y=6
===============
>> Processing line:
ID=1&X=8&Y=16 | ID=2&X=8&Y=18 | ID=3&X=7&Y=16 | ID=4&X=16&Y=16 | ID=5&X=11&Y
    =11 | ID=6&X=11&Y=7
===============
>> Processing line:
ID=1&X=8&Y=16 | ID=2&X=8&Y=18 | ID=3&X=8&Y=16 | ID=4&X=17&Y=16 | ID=5&X=11&Y
    =12 | ID=6&X=11&Y=8
===============
>> Processing line:
ID=1&X=8&Y=16 | ID=2&X=8&Y=18 | ID=3&X=8&Y=16 | ID=4&X=18&Y=16 | ID=5&X=11&Y
    =14 | ID=6&X=11&Y=9
===============
>> Processing line:
ID=1&X=9&Y=16 | ID=2&X=9&Y=18 | ID=3&X=8&Y=16 | ID=4&X=19&Y=16 | ID=5&X=11&Y
    =14 | ID=6&X=11&Y=10
===============
```

4.5.3.3 Running the Overlay Nodes

We execute an **Overlay Node Client Application** in the same way as in the first test scenario using the ID to specify the simulation data that the node fetches from the simulator.

For this test scenario we run an application for each of the six nodes in this scenario each with a different ID from 1 to 6 according to the simulation data.

When we run all the nodes applications we can see in its output the information of each group that corresponds to the responses from the **Group Server**:

Outputs of the Nodes when Joining the Overlay

- Output of the Nodes with IDs 1, 2 and 3

```
Group
GroupID=1
XMin=0
XMax=9
YMin=15
YMax=20
CentralNode
```

```
NodeID=1
X=8
Y=16
IP=127.0.0.1
Nodes
NodeID X Y IP
1 8 16 127.0.0.1
2 8 18 127.0.0.1
3 4 16 127.0.0.1

====================
```

- Output of the Nodes with IDs 5 and 6

```
Group
GroupID=2
XMin=9
XMax=13
YMin=0
YMax=14
CentralNode
NodeID=5
X=11
Y=6
IP=127.0.0.1
Nodes
NodeID X Y IP
5 11 5 127.0.0.1
6 11 1 127.0.0.1

================
```

- Output of the Node with ID 4

```
Group
GroupID=3
XMin=9
XMax=20
YMin=15
YMax=20
CentralNode
NodeID=4
X=11
Y=15
IP=127.0.0.1
Nodes
NodeID X Y IP
4 11 15 127.0.0.1
```

```
===============
```

Operation Output of Node with ID 1

Below we show the output the node with ID 1 in the ninth and tenth iterations of the simulation. The node with ID 1 is also the node with NodeID 1 and the **Central Node** of Group 1 at the ninth iteration. We see in the node's application output that the nodes that stop at the semaphores send the indicated broadcast message to their group and that in the update process, when the node's position leaves the group borders it gets assigned with a new group by the **Group Server**. So, this output demonstrates the update process of nodes and the exchange of messages between nodes with this overlay topology:

```
>> Request Host: http://localhost:14000
>> Request URI: /fetch?ID=1
<< Response: HTTP/1.1 200 OK
===============
X=8&Y=16>> Updating Node
>> Request Host: http://localhost:12000
>> Request URI: /updatenode?NodeID=1&X=8&Y=16
<< Response: HTTP/1.1 200 OK
===============
ThisNode
NodeID=1
Group
GroupID=1
XMin=0
XMax=9
YMin=15
YMax=20
CentralNode
NodeID=1
X=8
Y=16
IP=127.0.0.1
Nodes
NodeID X Y IP
1 8 16 127.0.0.1
2 8 18 127.0.0.1
3 7 16 127.0.0.1


===============
>> Updating Central Node in Group
>> Request Host: http://127.0.0.1:8002
>> Request URI: /update?NodeID=1&X=8&Y=16&IP=192.168.1.109
<< Response: HTTP/1.1 200 OK
===============
+OK Node Updated
```

```
>> Request Host: http://127.0.0.1:8003
>> Request URI: /update?NodeID=1&X=8&Y=16&IP=192.168.1.109
<< Response: HTTP/1.1 200 OK
===============
+OK Node Updated

===============
>> Request Host: http://127.0.0.1:8002
>> Request URI: /sendMsg?FROM=1&TO=2&MSG=SemaphoreIsRed!
<< Response: HTTP/1.1 200 OK
===============
>> Request Host: http://127.0.0.1:8003
>> Request URI: /sendMsg?FROM=1&TO=3&MSG=SemaphoreIsRed!
<< Response: HTTP/1.1 200 OK
===============
>>Updating Node
>> Request Host: http://127.0.0.1:8003
>> Request URI: /update?NodeID=2&X=8&Y=18&IP=127.0.0.1
<< Response: HTTP/1.1 200 OK
===============
+OK Node Updated

+OK Node Updated
===============
>>Processing Message...
>> Request Host: http://127.0.0.1:8003
>> Request URI: /sendMsg?FROM=2&TO=3&MSG=SemaphoreIsRed!
<< Response: HTTP/1.1 200 OK
===============
>> Received Broadcast Message: SemaphoreIsRed!
+OK BroadcastDelivered (-0)
===============
>>Updating Node
>> Request Host: http://127.0.0.1:8002
>> Request URI: /update?NodeID=3&X=7&Y=16&IP=127.0.0.1
<< Response: HTTP/1.1 200 OK
===============
+OK Node Updated

+OK Node Updated
===============
>>Processing Message...
>> Request Host: http://127.0.0.1:8002
>> Request URI: /sendMsg?FROM=3&TO=2&MSG=SemaphoreIsRed!
<< Response: HTTP/1.1 200 OK
===============
>> Received Broadcast Message: SemaphoreIsRed!
+OK BroadcastDelivered (-0)
```

```
==============

>> Request Host: http://localhost:14000
>> Request URI: /fetch?ID=1
<< Response: HTTP/1.1 200 OK
==============
X=9&Y=16>> Updating Node
>> Request Host: http://localhost:12000
>> Request URI: /updatenode?NodeID=1&X=9&Y=16
<< Response: HTTP/1.1 200 OK
==============
ThisNode
NodeID=1
Group
GroupID=3
XMin=9
XMax=20
YMin=15
YMax=20
CentralNode
NodeID=4
X=17
Y=16
IP=127.0.0.1
Nodes
NodeID X Y IP
4 18 16 127.0.0.1
1 9 16 127.0.0.1

==============
>> Getting New Central Node
>> Request Host: http://localhost:12000
>> Request URI: /operation?GroupID=1
<< Response: HTTP/1.1 200 OK
==============
>> Updating Nodes
>> Request Host: http://127.0.0.1:8002
>> Request URI: /leave?NodeID=1&X=8&Y=16&IP=192.168.1.109&NewCentralNodeID=3
<< Response: HTTP/1.1 200 OK
==============
>> Request Host: http://127.0.0.1:8003
>> Request URI: /leave?NodeID=1&X=8&Y=16&IP=192.168.1.109&NewCentralNodeID=3
<< Response: HTTP/1.1 200 OK
==============
==============
NodeID=3
>> Joining Group
>> Request Host: http://127.0.0.1:8004
>> Request URI: /join?NodeID=1&X=9&Y=16&IP=192.168.1.109
```

```
<< Response: HTTP/1.1 200 OK
===============
+OK Node Joined
```

The first thing the node does in this output is to get its position from the **Simulator** through a "/fetch" request. Having its position it sends an update request to the **Group Server** which responds with the node's actual group information. Since this node is the **Central Node** of group 1, then it sends an update request to all the nodes in the group.

Then we can observe that the node sends a broadcast message to the group 1 and since the node with ID 1 is the **Central Node** of its group, it sends the messages directly to the two other nodes in the group.

Next we see two operations regarding the node with NodeID 2: we can verify that the node receives an update request of the node with NodeID 2, updates the node in its own database and then routes the update request to the node with NodeID 3 and then we can observe that the node with NodeID 2 sent a broadcast message to the **Central Node** (node with NodeID 1) and the Central Node processes the message and sends it to the other node in the group, the node with NodeID 3. Afterwards we verify the same proceedings for the node with NodeID 3.

The node with NodeID 1 continues with its operation and gets a new position from the **Simulator**. The node now is assigned with a new group by the **Group Server**, so it asks the **Group Server**, through an "/operation" request which node is the new Central Node of group 1. The **Group Server** responds that the new **Central Node** of group 1 is the node with NodeID 3.

Finally the node with NodeID 1 leaves its group informing the other nodes in the group of the new **Central Node** and using the information received from the update request to the Group Server it joins the group 3 by sending a message to the **Central Node** of group 3, the node with NodeID 4.

4.5.3.4 Conclusions

We can verify through the output of each node application that the all the nodes position themselves in the correct groups and move to other groups according to the defined simulation data. We can also verify that the nodes that stop at the semaphores send the "SemophoreIsRed" broadcast message to their group overlay informing the other nodes in the group of the event.

With this scenario we proved that the framework supports different kinds of vehicle groupings based on their position.

4.6 Third Test Scenario

4.6.1 Test Specifications

4.6.1.1 Scenario Definition

This third test scenario is equal in every aspect to the first test scenario, except for the group overlay network topology. We have one group with a size of 2 x 20 in the used variables (X and Y) as we can see in the Figure 4.6 already showed in the first test scenario.

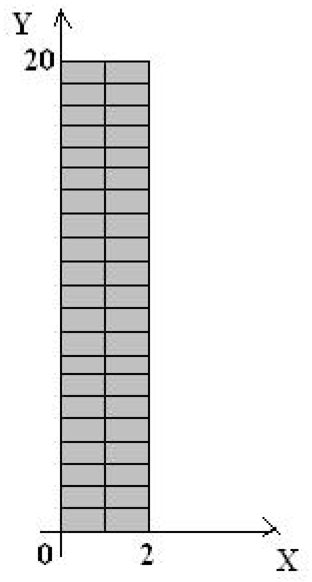

Figure 4.6: Road Group

Like in the first test scenario we use five vehicles as nodes. One of the nodes will have an accident and send a broadcast message to the group overlay.

In this scenario we want to test the framework with a random group overlay topology, so we created the following network topology:

Example Network Topology Implementation

In this example implementation of the **OverlayProcessor** and **HandlerProcessor** interfaces we use a different network topology from the default. We developed these interfaces implementation classes in order to also interact with the **DefaultServerHandlerProcessor** class that implements the **ServerHandlerProcessor** interface from the **Group Server** but without using the **Central Node** information.

It is not necessary to explain separately the proceedings in this network topology. The idea is that a node in a group overlay can only send requests to its neighbor nodes in the group and then the neighbor nodes themselves forward the requests to their neighbors. A node only has two neighbor nodes that are the nodes with the closest NodeID to its own NodeID. To prevent the messages from being eternally propagated in the network a **TTL** (Time To Live) mechanism was implemented and each node puts its NodeID in the request they send to other nodes. An example of the connection between nodes with this topology is represented in figure 4.7.

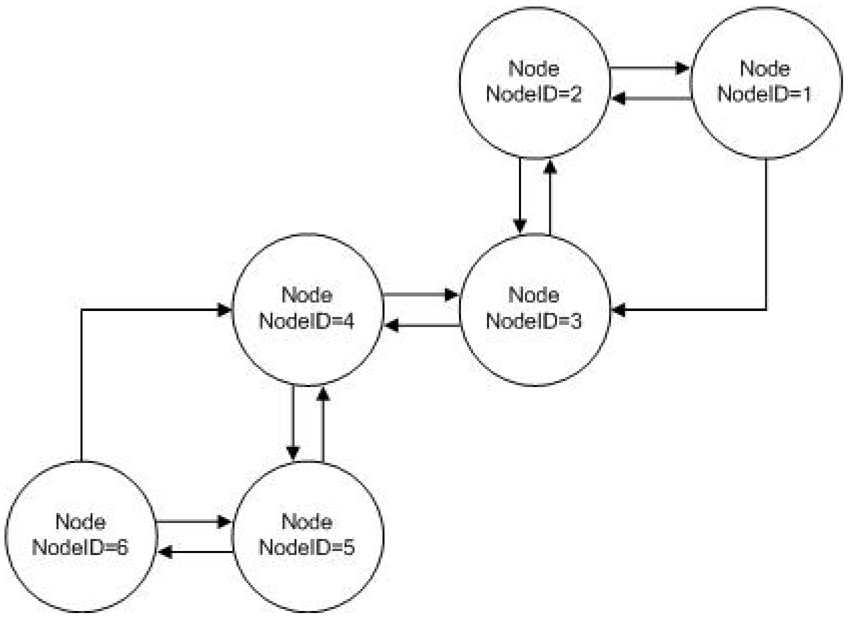

Figure 4.7: Example Network Topology with Six Nodes

- Joining Group Example

  To explain better how this mechanism works we explain how it works with a join group request. If we have already four nodes in a group with four nodes with NodeIDs from 1 to 4 and a node with the NodeID 5 wants to join the group it sends a join group request to its neighbors, the nodes with NodeIDs 3 and 4.

  Then since the node with NodeID 3 knows the received join group request has already been delivered to the node with NodeID 4 it forwards the join group request to the node with NodeID 2.

  Since the node with NodeID 4 also knows the node with NodeID 3 has received the request and the node that sent the request was the node with NodeID 5 it does not forward the request to any node.

  The node with NodeID 2 receives the join group request and it only sends to one of its neighbors, the node with NodeID 1 because it knows the request was delivered to it by the node with NodeID 3.

  Then the node with NodeID 1 knows the node that delivered the request also delivered the request to its neighbors and since the nodes with NodeID 1 and 2 share the same neighbor, the node with NodeID 3, the node with NodeID 1 does not forward the request to any node.

If for some reason any of the nodes forward the request to a node that already has received the same request, the **TTL** mechanism prevents the nodes from forward the same requests over and over again.

4.6.1.2 Scenario Conditions

The overlay nodes should get their position through a simulator using the **GetPositionFromSimulator** class each five seconds.

The simulation data will be on the already used text file named "first_test_scenario.txt" (see **First Test Scenario**) so the **Simulator** will use the **SimulatorDatabaseTextFile** class to get the simulation data. In the file we have the position for the five nodes in fifteen intervals of time of the defined five seconds.

When we write the **Overlay Node Client Application** we will also define that the node that has the accident is the node with ID equal to two.

Like in the first test scenario, in the **Group Server** we will use the default implementation class RoadGroups for creating the defined group and use the **DefaultServerHandlerProcessor** class to define the responses of the **Group Server**.

We now have to implement the defined overlay network topology using the **OverlayProcessor** and **HandlerProcessor** interfaces of the **Overlay Nodes**. We named this new classes as **TestOverlayProcessor** and **TestHandlerProcessor**.

We created another class, the **TestOverlayUtil** class that provides the **TestOverlayProcessor** and **TestHandlerProcessor** classes with the methods to find the neighbor nodes of the desired node as described in the network topology explanation.

TestOverlayProcessor

Each time a method from this class is called, the class calculates the TTL to add in the requests to send and discovers its neighbor nodes to which the requests will be sent. The TTL value is a function of the total number of nodes in the group overlay - N. With this network topology, a request that is sent by one node to a group overlay with N nodes reaches all nodes in N-2 hops. Since the TTL should be superior to the calculated number of hops we have that TTL = N. The two neighbor nodes are discovered also based in the NodeID like it was explained before. These nodes are added to the vector **neighborNodes** variable of this class.

- Process the response from the **Group Server** to an add node request

  After receiving the response from the **Group Server** to the add node request this method does the same as the one in the **DefaultOverlayProcessor** class to process response, except it does not use a **Central Node** so it discards the **Central Node** information in the response from the **Group Server**. The node has now all the information of its group overlay.

Afterwards the node starts its operation as an HTTP server calling the static method "NodeHTTPServer(HandlerProcessor pc)" from the **OverlayUtil** class using the **TestHandlerProcessor** class to define its HTTP server responses to the requests.

Finally the join group requests are sent to the neighbor nodes of the node that runs this method. These requests contain not only the node information but also the TTL and the NodeID of the sender node. So the format of the join request sent to the neighbor nodes is "/join?NodeID=int&X=int&Y=int&IP=String & TTL=int & SenderID=int".

- Leave the Group Overlay

  When a node wants to leave its group overlay it simply sends a leave group request to its neighbor nodes in order to inform all the nodes in the group. The leave group request has the following format: "/leave?NodeID=int&TTL=int&SenderID=int".

- Process the response from the **Group Server** to an update node request

  Like defined is this method of the **DefaultOverlayProcessor** class, when a node sends an update node request to the **Group Server** and receives its response it gets the assigned GroupID from it.

  If the new GroupID is equal to the actual GroupID of the static **Group** object, then the node sends an update node in group request to its neighbor nodes. The update node in group request format is similar to the join group request: "/update?NodeID=int&X=int&Y=int&IP=String&TTL=int&SenderID=int".

  But if the new GroupID is different from the actual GroupID, then the node uses the "groupLeaveNode()" method and then does the same proceedings as in the "processResponseAddNode(String response)" method using the response from the **Group Server** to join the new group overlay.

- Send Message

  When a node wants to send a message to another node in the group it runs this method. This method first codes the message to send so that all of the characters in the message are allowed in the HTTP request. Then the send message request containing the coded message is sent to the node's neighbors. All messages sent also use a unique identifier - MID instead of the sender node NodeID (SenderID). This message identifier is a long value and is calculated using hour and date in which the message was sent plus also a random integer number.

  So the format of the send message request is "/sendMsg?FROM=int &TO=int &MSG=String &TTL=int &MID=long".

- Send Broadcast Message

When a node wants to send a message to all the nodes in the group it uses this method. Like the send message method this method first codes the messages to send so that all of the characters in the message are allowed in the HTTP request. The format of the broadcast message request is the same as in the normal message request but with the "TO" camp value set as -1. Then node sends the broadcast message requests to its neighbor nodes.

TestHandlerProcessor

In all the methods of this class that process the requests the first thing they do is to get the TTL from the request using the static "getData(String input, type)" method from the **OverlayUtils** class with type equal to "TTL". The methods of this class that process a join group, an update node in group, and a leave group requests also get the sender node NodeID (SenderID) using the same method to get the TTL value but with type equal to "SenderID".

- Process a join group request

  When a node receives a join group request from another node it first gets the TTL value and the SenderID value as described before. Then it uses the static "getNode(String input, String IP, boolean aux)" method to get a node object of the node that sent the request and adds this node object to the vector **GroupNodes** of the static **Group** object.

  Finally it subtracts 1 to the TTL value and if the resulting TTL value is greater than zero and if the node is not an edge node (nodes in the group with the higher and the lower NodeIDs) then it sends the same join group request to its neighbor nodes with the updated value of TTL and the node NodeID as the "SenderID".

- Process an update node in group request

  When a node receives an update node in group request it operates the same way as when it receives a join group request, but instead of adding the received node to the **GroupNodes** vector, it updates the node in the vector.

- Process a leave group request

  Like in the previous two methods, when a node receives a leave group request, it first retrieves the value of TTL and SenderID from the request. Then it gets the NodeID of the node that is leaving identified in the request using the static "getData(String input, String type)" method with type equals to "NodeID". The method removes the node with the retrieved NodeID from the **GroupNodes** vector.

  Finally it subtracts 1 to the TTL value and if the resulting TTL value is greater than zero and if the node is not an edge node (nodes in the group with the higher and the lower NodeIDs) then it sends the same leave group request to its neighbor nodes with the updated value of TTL and its own NodeID as the "SenderID".

- Message handler

  When a node receives a send message request this method first gets the TTL value and instead of getting the SenderID value it gets the message identifier MID value from the request.

  Secondly it also gets the NodeID of the node that sent the message and the destination node NodeID from the request using the static "getData(String input, type)" method with type equals to "FROM" and "TO" respectively and also gets the message using the static "getDataString(String input, type)" method with type equals to "MSG".

  After getting all the necessary data from the request it verifies if message was already handled using the received message identification (MID). Each time a message is received this method saves the last received MID for each sender node. If the received MID is the same as the saved MID for the same sender node then the message was already received and process, so the method discards the message.

  The method then verifies if it received an individual message or a broadcast message. If the destination NodeID is higher than zero then it verifies if the node that received the request is the destination node comparing the retrieved destination NodeID and the node's NodeID and if it is the same the message is decoded and thus considered delivered. If the node is not the destination node it sends the received request to its neighbors altering the TTL value. If the destination NodeID is equal to -1 then it has received a broadcast message so it also sends the received request to its neighbor nodes altering the TTL value and it decodes the received message and considers the broadcast delivered.

Now that we have the required implementation classes we can write the applications.

4.6.2 Writting the Applications

For this scenario we use the same group and simulation data as in the first scenario so we will use the **Group Server Application** and **Simulator Application** written for the first test scenario. We only have to modify the topology of the **Overlay Node Client Application** of the first test scenario meaning we have to use the created **ExampleOverlayProcessor** class instead of the **DefaultOverlayProcessor** class.

Like in the other test scenarios we will do the tests in the same machine so the **Group Server** and the **Simulator** host is on "localhost". The **Group Server** will listen to port 12000 and **Simulator** will listen to port 14000.

We just need to perform the necessary modification to the **Overlay Node Client Application** of the first test scenario by calling the created **TestOverlayProcessor** instead of the **DefaultOverlayProcessor** used in the first test scenario.

4.6.3 Results and Conclusions

Since for this scenario we only one to test a different group overlay network topology we will use the already created **Group Server Application** and the **Simulator Application** for the first scenario ("groupserver.jar" and "simulator.jar" files). We only have to create the "nodeclient.jar" jar file for the **Overlay Node Client Application**.

Like in the first scenario we begin to execute the **Group Server Application** and the **Simulator Application** using the "java -jar" command. Then we run the newly created **Overlay Node Client Application** for each node in the simulation indicating their position through the ID input.

4.6.3.1 Running the Overlay Nodes

We execute an **Overlay Node Client Application** for each of the five nodes in the scenario each with a different ID from 1 to 5 in the ID order.

We verify that all the nodes successfully enter the network using the defined topology in the nodes output. Below we show the output of the nodes with ID 1, 2 and 3 when entering the group overlay.

Joining the Overlay

- Node with ID 1

```
>> Starting Overlay Node Client
>> Request Host: http://localhost:14000
>> Request URI: /fetch?ID=1
<< Response: HTTP/1.1 200 OK
================
X=1&Y=1>> Node Joining
>> Request Host: http://localhost:12000
>> Request URI: /addnode?X=1&Y=1
<< Response: HTTP/1.1 200 OK
================
ThisNode
NodeID=1
Group
GroupID=1
XMin=0
XMax=3
YMin=0
```

```
YMax=21
CentralNode
NodeID=1
X=1
Y=1
IP=127.0.0.1
Nodes
NodeID X Y IP
1 1 1 127.0.0.1

===============
>> Joining Group
>> Starting HTTP Server
===============
>>Node Joining
+OK Node Joined
===============
>>Node Joining
+OK Node Joined
===============
>>Node Joining
+OK Node Joined
===============
>>Node Joining
+OK Node Joined
===============
```

- Node with ID 2

```
>> Starting Overlay Node Client
>> Request Host: http://localhost:14000
>> Request URI: /fetch?ID=2
<< Response: HTTP/1.1 200 OK
===============
X=1&Y=3>> Node Joining
>> Request Host: http://localhost:12000
>> Request URI: /addnode?X=1&Y=3
<< Response: HTTP/1.1 200 OK
===============
ThisNode
NodeID=2
Group
GroupID=1
XMin=0
XMax=3
YMin=0
YMax=21
CentralNode
NodeID=1
X=1
```

```
Y=1
IP=127.0.0.1
Nodes
NodeID X Y IP
1 1 1 127.0.0.1
2 1 3 127.0.0.1

===============
>> Joining Group
>> Starting HTTP Server
>> Request Host: http://127.0.0.1:8001
>> Request URI: /join?NodeID=2&X=1&Y=3&IP=192.168.1.2&TTL=1&SenderID=2
<< Response: HTTP/1.1 200 OK
===============
+OK Node Joined

===============
>>Node Joining
+OK Node Joined
===============
>>Node Joining
>> Request Host: http://127.0.0.1:8001
>> Request URI: /join?NodeID=4&X=2&Y=11&IP=192.168.1.2&TTL=2&SenderID=2
<< Response: HTTP/1.1 200 OK
===============
+OK Node Joined
===============
>>Node Joining
>> Request Host: http://192.168.1.2:8001
>> Request URI: /join?NodeID=5&X=2&Y=16&IP=192.168.1.2&TTL=2&SenderID=2
<< Response: HTTP/1.1 200 OK
===============
+OK Node Joined
===============
```

- Node with ID 3

```
>> Starting Overlay Node Client
>> Request Host: http://localhost:14000
>> Request URI: /fetch?ID=3
<< Response: HTTP/1.1 200 OK
===============
X=1&Y=6>> Node Joining
>> Request Host: http://localhost:12000
>> Request URI: /addnode?X=1&Y=6
<< Response: HTTP/1.1 200 OK
===============
ThisNode
NodeID=3
Group
```

```
GroupID=1
XMin=0
XMax=3
YMin=0
YMax=21
CentralNode
NodeID=1
X=1
Y=1
IP=127.0.0.1
Nodes
NodeID X Y IP
1 1 1 127.0.0.1
2 1 3 127.0.0.1
3 1 6 127.0.0.1

===============
>> Joining Group
>> Starting HTTP Server
>> Request Host: http://127.0.0.1:8001
>> Request URI: /join?NodeID=3&X=1&Y=6&IP=192.168.1.2&TTL=2&SenderID=3
<< Response: HTTP/1.1 200 OK
===============
+OK Node Joined

>> Request Host: http://127.0.0.1:8002
>> Request URI: /join?NodeID=3&X=1&Y=6&IP=192.168.1.2&TTL=2&SenderID=3
<< Response: HTTP/1.1 200 OK
===============
+OK Node Joined

===============
>>Node Joining
+OK Node Joined
===============
>>Node Joining
>> Request Host: http://192.168.1.2:8002
>> Request URI: /join?NodeID=5&X=2&Y=16&IP=192.168.1.2&TTL=3&SenderID=3
<< Response: HTTP/1.1 200 OK
===============
+OK Node Joined
===============
```

Analyzing these outputs we verify that the first node to join the overlay, the node with ID 1 and the lowest NodeID receives the join requests corresponding to the four nodes that enter the group overlay and since this node has the lowest NodeID it does not need to send the request to its neighbor nodes.

The node with ID 2 enters the network in second and sends a join message to the node with ID 1. Then the node with ID 2 routes the join requests of the nodes that follow in joining the network to the node with ID 1 with the exception of the node with ID 3 join request, because the node with ID 3 sends the join request directly to the node with ID 1 since it is one of its neighbors when it joins the network.

On the other hand, the node with ID 3 receives the join requests of the other nodes that join the group overlay (nodes with ID 4 and 5). It does not route the join request of the node with ID 4 because the node with ID 4 sends its join request directly to the node with ID 2. So the node with ID 3 only routes the received join request from the node with ID 5 to the node with ID 2.

4.6.3.2 Conclusions

Observing the results we verify that the behavior of the nodes is according to the defined overlay network topology, so we conclude from this test scenario that the framework supports different types of network topologies.

4.7 Analysis

In this section we identify a number of features that are supported by the framework and why. We also analyse why the framework does not support a number of other features.

The test scenarios results and the framework architecture show us that the framework supports:

- Creation of static groups whose borders are limited by location coordinates;
- Attribution of the groups to nodes based on their position (a group is assigned to a node if that node's position is within the group borders). The overlay networks are established when the nodes connect themselves based on their group thus creating group overlays;
- Development of different overlay network topologies based on the nodes data (NodeID, node's position and last update in the overlay). The overlay network topology may support the identification of the nodes close to a specific node;
- Updating of nodes data in their group overlay network according to the defined network topology. In the update process if a node's new position is outside its group borders it lefts its current group overlay and establishes or connect with nodes in a new group overlay assigned to them based on its position.
- The exchange of messages between nodes in an established group overlay network (a node can send an individual message to other node in its group or broadcast messages to all the nodes in its group);

- Although we did not test the framework with a GPS device, the framework interfaces support obtaining coordinates from GPS devices because the position is get separately from the node's operation class using the **GetPosition** interface. The coordinates must be provided to the node's operation class, the **ClientOverlay** class, using the defined position format of X and Y. So we can get the coordinates through a GPS and convert them to a pair of integer values to provide these values to the node.

- The use of simulation data to provide the nodes with their position through the developed **Simulator** using outputs with simulation data saved on a text file. Although we did not use traffic algorithms and traffic simulators, it is also possible to use them because the simulation data is get through the **SimulatorDatabase** interface. So, to use traffic algorithms and traffic simulators we need to implement classes for the **SimulatorDatabase** interface.

With some changes to the classes and interfaces of the framework it would be possible to support the following features:

- Creation of dynamic vehicle groupings such as a group which follows a vehicle position and has a range of 1Km;

- Creation of groups based on the vehicles speed and direction;

- Identification of nodes in a group overlay using their speed or direction;

- Use of different overlay network topologies for different groups of one **Group Server** at the same time, since an **Overlay Node Client Application** is implemented in order to support only one overlay network topology and the nodes can exchange groups depending on their position, so the group overlays of one **Group Server** must all have the same network topology (limitation of an overlay network topology for all the groups in a **Group Server**).

Chapter 5

Conclusions

We have presented the ITS Overlay Framework that aims at facilitating the development and implementation of overlay networks for ITS. The ITS Overlay Framework has been designed with interfaces at various levels that provide the means to explore different types of vehicle groupings and different overlay network topologies.

We believe that the ITS Overlay Framework can be used in the research of Overlay Networks for Intelligent Transportation Systems and that it can also be used as an educational tool in the overlay networks area. The main benefits of this framework for the researchers and software developers include:

- providing an approach to the development and simulation of ITS overlays that eases the transition to testbeds and real-case usage;
- allowing users to create custom vehicle groupings based on coordinates each associated with a different overlay network;
- allowing users to define how nodes join a group overlay network based on their position, update themselves in the group overlay or switch to another group overlay, and send messages between themselves in the same group;
- supporting the development of different overlay network topologies and protocols
- performing simulation of ITS overlay networks for different scenarios using simulation data, algorithms or microscopic simulators;

The ITS Overlay Framework architecture is based on a common overlay architecture using a layered scheme. The benefits of this scheme is that developers can focus on a particular part of an ITS system without needing to worry about the other parts of the system; this scheme is supported by other frameworks such as the PlanetSim overlay simulation/experimentation framework.

Moreover, we used a generic centralized topology and an arbitrarily-defined topology to demonstrate that the framework allows the use of different overlay topologies.

Further investigation into how useful the framework is for developing ITS overlay applications would benefit from deploying the application in a real ITS scenario.

Future Work

As discussed in the **Analysis** section there are some features this framework does not support that would be important to implement in this framework's context.

It is possible to extend the framework to the not support features presented in the **Analysis** section by performing changes to some of the implemented classes and interfaces of the **Overlay Node** and the **Group Server**.

For the creation of dynamic vehicle groupings feature (e.g. a group which follows a vehicle position and has a determined radius) we should first modify the **Group** class to put a radius variable and a node variable. Then we must modify the **GroupsCreator** interface to get groups in function of the nodes. Finally we should change the **GroupServerDatabase** to support the created variables and we should also create a vector of groupsv́ariable in the **Overlay Node** since this feature allows nodes to have more than one group.

The creation of groups based on the vehicles speed and direction and identification of nodes in a group overlay using their speed or direction features can be made by changing the **Group** class adding the speed and direction variables and a method to get a group by speed and/or direction. Having this done we just need to change the requests and their responses.

As for the use of different overlay network topologies for different groups of one **Group Server** feature we first need to create a variable in the **Groups** class that indicates its overlay network topology and add this information to the responses of the **Group Server**. Then we should add a method to the **ClientOverlay** class of the **Overlay Node** that allows changing the implementation class of the **OverlayProcessor** interface according to the node's group.

Appendix A

Source Code

This appendix contains the source code of the most important classes and interfaces of the framework discussed in this thesis. The classes and interfaces source code is presented according to the thesis structure.

A.1 Framework Elements Classes

A.1.1 Node

```
package overlay.core.elements;

import java.sql.Timestamp;

public class Node {

        public int NodeID;
        public int x;
        public int y;
        public String IP;
        public Timestamp LastUpdate;
        public boolean Central = false;

        public int speed = -1;
        public int dir = 0;

        /**
        * Contructor of the Node class.
        *
        * @param int NodeID - NodeID of the node;
        * @param int x - X coordinate of the node;
        * @param int y - Y coordinate of the node;
        * @param String IP - IP of the node;
        */
        public Node(int NodeID, int x, int y, String IP){
                this.NodeID = NodeID;
                this.x = x;
                this.y = y;
                this.IP = IP;
        }

        /**
        * Sets this node's Central boolean as true.
        *
        */
        public void Central(){
                this.Central = true;
        }

        /**
        * Sets this node's Central boolean as false.
        *
        */
        public void NotCentral(){
                this.Central = false;
        }

        /**
        * Updates the timestamp LastUpdate variable.
        *
        * @param Timestamp time - New value for the LastUpdate variable;
        */
        public void setLastUpdate(Timestamp time){
                this.LastUpdate = time;
        }

        /**
        * Change the NodeID of a node to the provided NodeID.
        *
        * @param int NodeID - New value of NodeID of the node;
        */
        public void setNodeID(int NodeID){
                this.NodeID = NodeID;
        }

        /**
        * This method allows a Node object to have its values updated.
        *
        * @param int NodeID - NodeID of the node;
        * @param int x - X coordinate of the node;
        * @param int y - Y coordinate of the node;
        * @param String IP - IP of the node;
        */
        public void updateNode(int NodeID, int x, int y, String IP){
                this.NodeID = NodeID;
                this.x = x;
                this.y = y;
                this.IP = IP;
        }

        /**
        * Creates a string with the node's information.
        */
        public String toString(){
                String temp;
                temp = NodeID + "␣" + x + "␣"  + y + "␣" + IP;
                return temp;
        }

        /**
        * Creates a string containing an URI with node information to use in
              requests.
        */
        public String toURI(){
                String temp;
                temp = "NodeID="+NodeID+"&X="+x+"&Y="+y+"&IP="+IP;
                return temp;
        }

        /**
        * Creates a string of the actual node as the central node to use in
             the Group Server.
        */
        public String toStringCentral(){
                String temp;
                temp = "CentralNode\r\nNodeID="+NodeID+"\r\nX="+x+"\r\nY="+y
                    +"\r\nIP="+IP;
                return temp;
        }
}
```

A.1.2 Group

```
package overlay.core.elements;

import java.sql.Timestamp;
import java.util.Vector;


public class Group {

        public int GroupID = −1;
        public Vector<Node> GroupNodes;
        public Node CentralNode = null;
        public Node ThisNode = null;
public int xmin = -1;
public int xmax = -1;
public int ymin = -1;
public int ymax = -1;

/**
* Creates a group with the provided GroupID.
*
* @param int GroupID of the assigned group;
*/
public Group (int GroupID){
        this.GroupID = GroupID;
        GroupNodes = new Vector<Node>();
}

/**
* Creates a group with the provided parameters.
*
* @param int GroupID of the assigned group;
* @param int xmin
* @param int xmax
* @param int ymin
* @param int ymax
*/
public Group (int GroupID, int xmin, int xmax, int ymin, int ymax){
        this.GroupID = GroupID;
        GroupNodes = new Vector<Node>();
        this.xmin = xmin;
        this.xmax = xmax;
        this.ymin = ymin;
        this.ymax = ymax;
}

/**
* Adds the provided node to the group.
* If the central boolean of the node is true then it is also the
   central node.
*
* @param Node node to be added to the group;
*/
public void addNode(Node node){
        if(node.Central == false)
                GroupNodes.add(node);
        else
                CentralNode = node;
}

/**
```

```
* Method to update the node with the provided NodeID with the
   provided newnode.
*
* @param int NodeID of the node to be updated;
* @param Node newnode − Updated node;
*/
public void updateNode(int NodeID, Node newnode){
        for(int i=0; i<GroupNodes.size(); i++){
                Node temp = GroupNodes.get(i);
                if(temp.NodeID == NodeID){
                        GroupNodes.set(i, newnode);
                        break;
                }
        }
}

/**
* Removes the provided node from the group.
*
* @param Node node to be removed;
*/
public void delNode(Node node){
        GroupNodes.remove(node);
        if(CentralNode.NodeID == node.NodeID){
                CentralNode = null;
        }
}

/**
* Sets the provided node as This Node.
*
* @param Node node containing the node's information;
*/
public void defineThisNode(Node node){
        ThisNode = node;
}

/**
* Builds a string containing the group and its nodes information.
*/
public String toString(){
        StringBuffer response = new StringBuffer();
        response.append("Group\r\nGroupID="+GroupID+"\r\n");
        if(xmin!=−1 && xmax!=−1 && ymin!=−1 && ymax!=−1){
                response.append("XMin="+xmin+"\r\n");
                response.append("XMax="+xmax+"\r\n");
                response.append("YMin="+ymin+"\r\n");
                response.append("YMax="+ymax+"\r\n");
        }
        if(CentralNode!=null)
                response.append(CentralNode.toStringCentral()+"\r\n"
                    );
        if(ThisNode != null){
                response.append("ThisNode\r\n");
                response.append(ThisNode.toString()+"\r\n");
        }
        if(GroupNodes.size()>0){
                response.append("Nodes\r\n");
                response.append("NodeID␣X␣Y␣IP\r\n");
                for(int i=0; i<GroupNodes.size(); i++){
                        response.append(GroupNodes.get(i).toString()
                            +"\r\n");
                }
        }
        return response.toString();
}

/**
* Method to get the node in the group with the provided NodeID.
*
* @param int NodeID of the desired node;
* @return Node with the provided NodeID;
*/
public Node getNodeByID(int NodeID){
        if(ThisNode != null)
                if(ThisNode.NodeID==NodeID)
                        return ThisNode;
        for(int i=0; i<GroupNodes.size(); i++){
                        Node temp = GroupNodes.get(i);
                        if(temp.NodeID == NodeID)
                                return temp;
        }
        return null;
}

/**
* Method to get the last updated node.
*
* @return Node - Last updated node
*/
public Node getLastUpdatedNode(){
        int NodeID = 0;
        Timestamp time = new Timestamp(0);
        for(int i=0; i<GroupNodes.size(); i++){
                Node temp = GroupNodes.get(i);
                if(temp.LastUpdate.after(time)){
                        time = temp.LastUpdate;
                        NodeID = temp.NodeID;
                }
            }
            return getNodeByID(NodeID);
        }

        /**
        * Method to get the minimum NodeID in the group.
        *
        * @return int The minimum NodeID in the group;
        */
        public int getMinNodeID(){
            int MinNodeID=ThisNode.NodeID;
            for(int i=0; i<GroupNodes.size(); i++){
                if(GroupNodes.get(i).NodeID < MinNodeID)
                    MinNodeID = GroupNodes.get(i).NodeID;
            }
            return MinNodeID;
        }

        /**
        * Method to get the maximum NodeID in the group.
        *
        * @return int The maximum NodeID in the group;
        */
        public int getMaxNodeID(){
            int MaxNodeID=ThisNode.NodeID;
            for(int i=0; i<GroupNodes.size(); i++){
                if(GroupNodes.get(i).NodeID > MaxNodeID)
                    MaxNodeID = GroupNodes.get(i).NodeID;
            }
            return MaxNodeID;
        }

}
```

A.2 Network Communications Layer

A.2.1 Group Server Classes

A.2.1.1 GroupServerHTTPServer

```
package overlay.server.com;

import java.io.IOException;
import java.net.InetSocketAddress;

import overlay.server.com.handlers.AddNodeHandler;
import overlay.server.com.handlers.DelNodeHandler;
import overlay.server.com.handlers.Operation;
import overlay.server.com.handlers.UpdateNodeHandler;
import overlay.server.com.handlers.processor.ServerHandlerProcessor;

import com.sun.net.httpserver.HttpServer;

public class GroupServerHTTPServer {

        public int PORT = 10000;
        public static final int BACKLOG = 0;
        public static final String URL_CONTEXT_DEL = "/delnode";
        public static final String URL_CONTEXT_ADD = "/addnode";
        public static final String URL_CONTEXT_UPDATE = "/updatenode";
        public static final String URL_CONTEXT_OPERATION = "/operation";

        public GroupServerHTTPServer(int Port, ServerHandlerProcessor pc)
            throws IOException{
                PORT = Port;
                System.out.println(">> Starting HTTP Server");
                this.startHTMLServer(pc);
        }

        private void startHTMLServer(ServerHandlerProcessor pc) throws
            IOException{
              final HttpServer server =
                  HttpServer.create(new InetSocketAddress(PORT), BACKLOG);
               server.createContext(URL_CONTEXT_DEL, new DelNodeHandler(pc))
                   ;
               server.createContext(URL_CONTEXT_ADD, new AddNodeHandler(pc))
                   ;
               server.createContext(URL_CONTEXT_UPDATE, new
                   UpdateNodeHandler(pc));
               server.createContext(URL_CONTEXT_OPERATION, new Operation(pc)
                   );
               server.setExecutor(null);
               server.start();
        }

}
```

A.2.2 Overlay Nodes Classes

A.2.2.1 Communication

```
package overlay.com;

import overlay.core.elements.Node;

public class Communication {

        public ElementalHttpPost post;
        public ElementalHttpPost post_server;
        private String server_host;
        private int server_port;

        public Communication(String host, int port){
                server_host = host;
                server_port = port;
                post_server = new ElementalHttpPost(server_host, server_port
                    );
        }

        public void setNewServer(String host, int port){
                server_host = host;
                server_port = port;
                post_server = new ElementalHttpPost(server_host, server_port
                    );
        }

        public String sendToServer(String URI){
                String response = "-ERR_Connection_Failed";
                try {
                        response = post_server.Post(URI);
                } catch (Exception e) {
                        e.printStackTrace();
                }
                return response;
        }

        private void changeHost(String host, int port){
                post = new ElementalHttpPost(host, port);
        }

        public String sendToNode(Node node, String URI){
                changeHost(node.IP, 8000 + node.NodeID);
                String response = "-ERR_Connection_Failed";
                try {
                        response = post.Post(URI);
                } catch (Exception e) {
                        e.printStackTrace();
                }
                return response;
        }
}
```

A.2.2.2 GroupServerRequests

```
package overlay.com.requests;

import overlay.core.ClientOverlay;

public class GroupServerRequests {
        public static String AddNode(int x, int y){
                String URI = "/addnode?X=" + x + "&Y="+ y;
                return ClientOverlay.Com.sendToServer(URI);
        }

        public static String UpdateNode(int x, int y){
                String URI = "/updatenode?NodeID="+ ClientOverlay.Group.
                    ThisNode.NodeID + "&X=" + x + "&Y="+ y;
                return ClientOverlay.Com.sendToServer(URI);
        }

        public static String LeaveNode(){
                String URI = "/delnode?NodeID="+ ClientOverlay.Group.
                    ThisNode.NodeID + "&GroupID=" + ClientOverlay.Group.
                    GroupID;
                return ClientOverlay.Com.sendToServer(URI);
        }

        public static String Operation(){
                String URI = "/operation?GroupID=" + ClientOverlay.Group.
                    GroupID;
                return ClientOverlay.Com.sendToServer(URI);
        }

}
```

A.2.2.3 OverlayRequests

```
package overlay.com.requests;

import overlay.core.ClientOverlay;
import overlay.core.elements.Node;

public class OverlayRequests {

        public static String join(Node DestNode, Node JoinNode, String
            Headers){
                return transmitNode(JoinNode, DestNode, "join", Headers);
                                }

        public static String leave(Node DestNode, Node LeaveNode, String
            Headers){
                return transmitNode(LeaveNode, DestNode, "leave", Headers);
        }

        public static String update(Node DestNode, Node UpdateNode, String
            Headers){
                return transmitNode(UpdateNode, DestNode, "update", Headers)
                    ;
        }

        public static String sendMsg(Node DestNode, int SenderID, int DestID
            , String message, String Headers){
                String URI = "/sendMsg?FROM="+SenderID+"&TO="+DestID+"&MSG="
                    +message+Headers;
                return ClientOverlay.Com.sendToNode(DestNode, URI);
        }

        public static String transmitNode(Node node, Node destnode, String
            type, String Headers){
                String URI = "/" + type + "?"+node.toURI()+Headers;
                return ClientOverlay.Com.sendToNode(destnode, URI);
        }

}
```

A.2.2.4 NodeHTTPServer

```
package overlay.com.servers;

import java.io.IOException;
import java.net.InetSocketAddress;

import overlay.com.handlers.GeneralJoinGroupHandler;
import overlay.com.handlers.GeneralLeaveGroupHandler;
import overlay.com.handlers.GeneralMessageHandler;
import overlay.com.handlers.GeneralUpdateGroupHandler;
import overlay.com.handlers.processor.HandlerProcessor;

import com.sun.net.httpserver.HttpServer;

public class NodeHTTPServer {

                public int PORT = 8000;
                public static final int BACKLOG = 0;
                public static final String URL_CONTEXT_LEAVE = "/leave";
                public static final String URL_CONTEXT_JOIN = "/join";
                public static final String URL_CONTEXT_UPDATE = "/update";
                public static final String URL_CONTEXT_SENDMSG = "/sendMsg";

                public NodeHTTPServer(int NodeID, HandlerProcessor pc)
                    throws IOException{
                        PORT = PORT + NodeID;
                        System.out.println(">>␣Starting␣HTTP␣Server");
                        this.startHTMLServer(pc);
                }

                private void startHTMLServer(HandlerProcessor pc) throws
                    IOException{
                        final HttpServer server = HttpServer.create(new
                            InetSocketAddress(PORT), BACKLOG);
                        server.createContext(URL_CONTEXT_LEAVE, new
                            GeneralLeaveGroupHandler(pc));
                        server.createContext(URL_CONTEXT_JOIN, new
                            GeneralJoinGroupHandler(pc));
                        server.createContext(URL_CONTEXT_UPDATE, new
                            GeneralUpdateGroupHandler(pc));
                        server.createContext(URL_CONTEXT_SENDMSG, new
                            GeneralMessageHandler(pc));
                        server.setExecutor(null);
                        server.start();
                }
}
```

A.2.3 Simulator Classes

A.2.3.1 SimulatorHTTPServer

```
package overlay.simulator.com;

import java.io.IOException;
import java.net.InetSocketAddress;

import overlay.simulator.core.SimulatorDatabase;

import com.sun.net.httpserver.HttpServer;

public class SimulatorHTTPServer {

        public int PORT = 7000;
        public static final int BACKLOG = 0;
        public static final String URL_CONTEXT = "/fetch";

        public SimulatorHTTPServer(SimulatorDatabase db, int PORT) throws
            IOException{
                System.out.println(">> Starting HTTP Server");
                this.PORT = PORT;
                this.startHTMLServer(db);
        }

        private void startHTMLServer(SimulatorDatabase db) throws
            IOException{
              final HttpServer server =
                  HttpServer.create(new InetSocketAddress(PORT), BACKLOG);
               server.createContext(URL_CONTEXT, new SimulatorHandler(db));
               server.setExecutor(null);
               server.start();
        }
}
```

A.3 Management Layer

A.3.1 Group Server Classes and Interfaces

A.3.1.1 ServerHandlerProcessor

```
package overlay.server.com.handlers.processor;

public interface ServerHandlerProcessor {

            /**
            * Method to process the "/addnode" request from a node by giving
                it a NodeID and assigning it to a group and then finally
                return a response the node.
            *
            * @param String request from a node;
            * @param String IP of the node that sent the request;
            * @return String that contains node and group information;
            */
            public String addNode(String request, String IP);

            /**
            * Method to process the "/updatenode" request from a node
                verifying if the node has leaved a group and entered other and
                 then return a response to it.
            *
            * @param String request from a node;
            * @param String IP of the node that sent the request;
            * @return String that contains node and group information;
            */
            public String updateNode(String request, String IP);

            /**
            * This method processes the "/delnode" request and is in charge
                of removing the node form the server database.
            *
            * @param String request from a node;
            * @return String informing if the node was successfully deleted;
            */
            public String delNode(String request);

            /**
            *
            *
            * @param String request from a node;
        * @return String with the desired response to the operation request;
            */
            public String operation(String request);
}
```

A.3.1.2 GroupServerDatabase

```
package overlay.server.core;

import java.sql.Timestamp;
import java.util.Vector;

import overlay.server.core.elements.Group;
import overlay.server.core.elements.Node;
import overlay.server.core.groups.GroupsCreator;

public class GroupServerDatabase {

        public Vector<Group> Groups;
        public int CurrentNodeID = 0;

        /**
        * Constructor of the GroupServerDatabase class.
        * Creates the groups defined by the provided implementation class of
             the GroupsCreator interface.
        *
        * @param GroupsCreator gc - Implementation class of the
            GroupsCreator interface;
        */
        public GroupServerDatabase(GroupsCreator gc){
                Groups = gc.getGroups();
                printGroups();
        }

        /**
        * Method to print in the console the saved groups information.
        */
        private void printGroups(){
                for(int i=0; i<Groups.size(); i++){
                        System.out.println(Groups.get(i).toString());
                }
        }

        /**
        * Method that returns the Group with the provided GroupID.
        *
        * @param int GroupID of the group;
        * @return Group with the provided GroupID;
        */
        public Group getGroupByID(int GroupID){
                Group temp = null;
                for(int i=0; i<Groups.size(); i++){
                        if(Groups.get(i).GroupID==GroupID)
                                temp=Groups.get(i);
        }
        return temp;
}

/**
* Method that returns the index of the group in the vector that
    contains the provided coordinates.
*
* @param int x - X coordinate;
* @param int y - Y coordinate;
* @return int Index of the group of the vector that contains the
    provided coordinates;
*/
public int getGroupIndByXY(int x, int y){
        for(int i=0; i<Groups.size(); i++){
                Group tempGroup = Groups.get(i);
                if(x >= tempGroup.xmin && x < tempGroup.xmax &&
                                y >= tempGroup.ymin && y <
                                    tempGroup.ymax)
                        return i;
        }
        return -1;
}

/**
* Method that returns the Group object that contains the node with
    the provided NodeID.
*
* @param int NodeID - NodeID of the node that is in the desired
    group;
* @return Group that contains the node with the provided NodeID;
*/
public Group getGroupByNodeID(int NodeID){
        Group temp = null;
        System.out.println("OLA1");
        for(int i=0; i<Groups.size(); i++){
                if(Groups.get(i).getNodeByID(NodeID)!=null){
                        temp=Groups.get(i);
                        System.out.println("OLA2");
                        break;
                }
        }
        return temp;
}

/**
* Method to replace the group saved in the vector with the provided
    GroupID with the provided Group.
*
```

```
* @param int GroupID - GroupID of the group that we want to replace;
* @param Group sub - Group that replaces the group with the provided
    GroupID;
*/
public void setGroupByID(int GroupID, Group sub){
        for(int i=0; i<Groups.size(); i++){
                if(Groups.get(i).GroupID==GroupID)
                        Groups.set(i, sub);
        }
}

/**
* Method that adds a node to the respective group.
*
* @param Node - node to be added;
* @return int GroupID of the assigned group;
*/
public int addNode(Node node){
        CurrentNodeID++;
        Timestamp date = new Timestamp(0);
        date.setTime(System.currentTimeMillis());
        node.setNodeID(CurrentNodeID);
        node.setLastUpdate(date);
        int GroupID = 0;
        int ind = getGroupIndByXY(node.x, node.y);
        if(ind<0)
                return GroupID;
        GroupID = Groups.get(ind).GroupID;
        for(int i=0; i<Groups.size(); i++){
                Group temp = Groups.get(i);
                if(temp.GroupID==GroupID){
                        if(temp.CentralNode == null)
                                temp.CentralNode = node;
                        temp.addNode(node);
                        Groups.set(i, temp);
                }
        }
        return GroupID;
}

/**
* Method that updates the node with the NodeID of the provided node
    object with the provided node object.
*
* @param Node - node to be updated;
* @return int GroupID of the assigned group;
*/
public int updateNode(Node node){
        Timestamp date = new Timestamp(0);
```

```
            date.setTime(System.currentTimeMillis());
            node.setLastUpdate(date);
            Group oldGroup = getGroupByNodeID(node.NodeID);
            int newGroupInd = this.getGroupIndByXY(node.x, node.y);
            if(newGroupInd<0)
                    return -1;
            Group newGroup = Groups.get(newGroupInd);
            if(oldGroup.GroupID == newGroup.GroupID)
                    for(int i=0; i<Groups.size(); i++){
                            Group temp = Groups.get(i);
                            if(temp.GroupID==newGroup.GroupID){
                                    temp.updateNode(node.NodeID, node);
                                    Groups.set(i, temp);
                            }
                    }
            else
                    for(int i=0; i<Groups.size(); i++){
                            Group temp = Groups.get(i);
                            if(temp.GroupID==oldGroup.GroupID){
                                    temp.delNode(temp.getNodeByID(node.
                                        NodeID));
                                    Groups.set(i, temp);
                            }
                            if(temp.GroupID==newGroup.GroupID){
                                    temp.addNode(node);
                                    if(temp.CentralNode==null)
                                            temp.CentralNode = node;
                                    Groups.set(i, temp);
                            }
                    }
            return newGroup.GroupID;
    }

    /**
    * Method that removes the node with the provided NodeID from the
        group with the provided GroupID.
    *
    * @param int NodeID of the node to be removed;
    * @param int GroupID of the group from whom the node will be removed
        ;
    */
    public void delNode(int NodeID, int GroupID){
            for(int i=0; i<GroupServer.dbase.Groups.size(); i++){
               Group group = GroupServer.dbase.Groups.get(i);
               if(group.GroupID == GroupID){
                       group.delNode(group.getNodeByID(NodeID));
                       GroupServer.dbase.Groups.set(i, group);
               }
            }
        }
}
```

A.3.2 Overlay Nodes Interfaces

A.3.2.1 OverlayProcessor

```
package overlay.core.processor;

public interface OverlayProcessor {

        /**
        * Change the value of the position on the node;
        *
        * @param int Value of X;
        * @param int Value of Y;
        */
        public void changeXY(int x, int y);

        /**
        * Process a response from the group server to join the indicated
            group.
        *
        * @param String Response from the group server;
        * @return int Value that indicates if the node joined the group;
        */
        public int processResponseAddNode(String response);

        /**
        * Process a response from the group server to update a node in a
            group.
        *
        * @param String Response from the group server;
        * @return int Value that indicates if the node uptated itself in the
             group;
        */
        public int processResponseUpdateNode(String response);

        /**
        * Method to inform the group that the node is leaving the group.
        *
        * @return String Indicates if the node sucessfully leaved the group;
        */
        public String groupLeaveNode();

        /**
        * Method to send a message to a node in the group.
        *
        * @param String Message to be send;
        * @param int NodeID from the destination node;
        * @return String saying if the message was sucessfuly delivered;
        */
        public String sendMsg(String message, int NodeID);

        /**
        * Method to send a broadcast message to all nodes in the group.
        *
        * @param String Message to be send;
        * @return String saying if the message was sucessfuly delivered;
        */
        public String sendBroadcast(String message);
}
```

A.3.2.2 HandlerProcessor

```
package overlay.com.handlers.processor;

public interface HandlerProcessor {

        /**
        * Method to process a join group request.
        *
        * @return String Response to the node that is trying to join the
            group;
        */
        public String joinGroup(String input, String IP);

        /**
        * Method to process a update node in group request.
        *
        * @return String Response to the node that is trying to update
            itself in the group;
        */
        public String updateGroup(String input, String IP);

        /**
        * Method to process a leave group request.
        *
        * @return String Response to the node that is trying to leave the
            group;
        */
        public String leaveGroup(String input);

        /**
        * Method to process the received messages from a node.
        *
        * @return String Response to the node that has send the message;
        */
        public String messageHandler(String request);
}
```

A.3.3 Simulator Interfaces

A.3.3.1 SimulatorDatabase

```
package overlay.simulator.core;

public interface SimulatorDatabase {

        /**
        * Gets the next position given by X and Y.
        */
        public void fetchXY();

        /**
        * Gives back the actual value of Y of a Node.
        *
        * @param int ID - Identification of the node;
        * @return int Y - Actual value of Y;
        */
        public int getX(int ID);

        /**
        * Gives back the actual value of Y of a Node.
        *
        * @param int ID - Identification of the node;
        * @return int Y - Actual value of Y;
        */
        public int getY(int ID);

        /**
        * Function to see if there is more position values.
        *
        * @return true  If there is not more position values to fecth;
        * @return false Otherwise;
        */
        public boolean isEnd();

        /**
        * Function to force the end boolean to be true in order to end the
            simulation.
        */
        public void setEnd();

        /**
        * Build a String to return via an HTTP response.
        *
        * @return String for HTTP response.
        */
        public String buildResponse(String input);
}
```

A.4 Application-Level Layer

A.4.1 Group Server Classes and Interfaces

A.4.1.1 GroupsCreator

```
package overlay.server.core.groups;

import java.util.Vector;
import overlay.server.core.elements.Group;

public interface GroupsCreator {

        /**
        * Method that creates the groups for the overlays.
        *
        * @return Vector<Group> Containing the groups;
        */
        public Vector<Group> getGroups();

}
```

A.4.1.2 GroupServer

```
package overlay.server.core;

import java.io.IOException;
import overlay.server.com.GroupServerHTTPServer;
import overlay.server.com.handlers.processor.ServerHandlerProcessor;
import overlay.server.core.groups.GroupsCreator;


public class GroupServer {

        public static GroupServerDatabase dbase;

        /**
        * Constructor of the GroupServer class.
        *
        * @param GroupsCreator gc - Implementation class of the
            GroupsCreator interface;
        * @param ServerHandlerProcessor pc - Implementation class of the
            ServerHandlerProcessor interface;
        * @param int port - Value of the port to which the Group Server HTTP
             server will listen;
        * @throws IOException
        */
        public GroupServer(GroupsCreator gc, ServerHandlerProcessor pc, int
            port) throws IOException{
                GroupServer.dbase = new GroupServerDatabase(gc);
                new GroupServerHTTPServer(port, pc);
        }

        /**
        * Method to change the implementation class of the GroupsCreator
            interface it uses in order to define different groups.
        *
        * @param GroupsCreator gc - New implementation class of the
            GroupsCreator interface;
        * @return String that containing the defined groups;
        */
        public String setGroups(GroupsCreator gc){
                GroupServer.dbase = new GroupServerDatabase(gc);
                return dbase.Groups.toString();
        }
}
```

A.4.2 Overlay Nodes Classes and Interfaces

A.4.2.1 ClientOverlay

```
package overlay.core;

import overlay.com.Communication;
import overlay.com.requests.GroupServerRequests;

import overlay.core.elements.Group;
import overlay.core.processor.OverlayProcessor;

public class ClientOverlay{

        public static Group Group;
        public static Communication Com;
        public OverlayProcessor pc;

        /**
        * Constructor of the class ClientOverlay.
        *
        * @param String host - Host address of the Group Server;
        * @param int port - Port of the Group Server;
        * @param OverlayProcessor pc - Implementation class of the
            OverlayProcessor interface;
        */
        public ClientOverlay(String host, int port, OverlayProcessor pc){
                ClientOverlay.Com = new Communication(host, port);
                this.pc = pc;
        }

        /**
        * Method to send a request to add the node to
* the Group Server and then process its response.
*
* @param int x - Value of position in the X axis.
* @param int y - Value of position in the Y axis.
*/
public int addNode(int x, int y){
        System.out.println(">> Node Joining");
        String response = GroupServerRequests.AddNode(x, y);
        System.out.println(response);
        System.out.println("==============");
        pc.changeXY(x, y);
        return pc.processResponseAddNode(response);
}

/**
* Method to send a request to update the node to
* the Group Server and then process its response.
*
* @param int x - Updated value of position in the X axis.
* @param int y - Updated value of position in the Y axis.
*/
public int updateNode(int x, int y){
        System.out.println(">> Updating Node");
        String response = GroupServerRequests.UpdateNode(x, y);
        System.out.println(response);
        System.out.println("==============");
        pc.changeXY(x, y);
        return pc.processResponseUpdateNode(response);
}

/**
* Method to inform the group overlay that the node is leaving and
   then
* send a request to the Group Server to remove the node.
*
*/
public String leaveNode(){
        System.out.println(">> Node Leaving");
        pc.groupLeaveNode();
        return GroupServerRequests.LeaveNode();
}

/**
* Method to send a message to a node in the
* same group overlay with the destination NodeID.
*
*/
public String sendMsg(String message, int NodeID){
        return pc.sendMsg(message, NodeID);
        }

        /**
        * Method to send a broadcast message to all
        * the nodes in the same group overlay.
        *
        */
        public String sendBroadcast(String message){
                return pc.sendBroadcast(message);
        }
}
```

A.4.2.2 GetPosition

```
package overlay.core.position;

public interface GetPosition {

        /**
        * Fetches the next pair of values X and Y.
        *
        */
        public void fetchXY();

        /**
        * Gets the value of X for the node.
        *
        * @return int Value of position in X;
        */
        public int getX();

        /**
        * Gets the value of Y for the node.
        *
        * @return int Value of position in Y;
        */
        public int getY();

}
```

A.4.3 Simulator Classes

A.4.3.1 SimulatorProcessor

```
package overlay.simulator.core;

import java.io.IOException;

import overlay.simulator.com.SimulatorHTTPServer;
```

```
public class SimulatorProcessor {

        public static SimulatorDatabase db;

        public SimulatorProcessor(SimulatorDatabase db, int PORT){
                SimulatorProcessor.db = db;
        try {
                        new SimulatorHTTPServer(db, PORT);
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }

        public void fetchXY(){
                db.fetchXY();
        }

        public void setEnd(){
                db.setEnd();
        }

        public boolean isEnd(){
                return db.isEnd();
        }
}
```

A.5 Default Implementation Classes

A.5.1 Group Server Classes

A.5.1.1 RoadGroups

```
package overlay.server.core.groups;

import java.util.Vector;
import overlay.server.core.elements.Group;

public class RoadGroups implements GroupsCreator {

        private int xSize;
        private int ySize;
        private int numHGroups;
        private int numVGroups;

        /**
        * Construtor of the class RoadGroups.
        *
        * @param int Number of horizontal groups;
        * @param int Number of vertical groups;
        * @param int Size of the groups in the X axis;
```

```
	* @param int Size of the groups in the Y axis;
	*/
	public RoadGroups(int numHGroups, int numVGroups, int xSize, int
	    ySize){
		this.numHGroups = numHGroups;
		this.numVGroups = numVGroups;
		this.xSize = xSize;
		this.ySize = ySize;
	}

	/**
	* Method that creates the groups for the overlays.
	*
	* @return Vector<Group> Containing the groups;
	*/
	public Vector<Group> getGroups() {
		return getRoadGroups();
	}


	/**
	* Method to create road groups.
	*
	* @return Vector<Group> Containing the road groups;
	*/
	private Vector<Group> getRoadGroups(){
		Vector<Group> Groups = new Vector<Group>();
		int xmin = 0;
		int ymin = 0;
		int CurrentGroupID = 1;
		for(int i=0; i<numVGroups; i++){
			Groups.add(new Group(CurrentGroupID, xmin, xmin+
			    xSize, ymin, ymin+ySize));
			CurrentGroupID++;
			for(int j=1; j<numHGroups; j++){
				xmin = xmin + xSize;
				Groups.add(new Group(CurrentGroupID, xmin,
				    xmin+xSize, ymin, ymin+ySize));
				CurrentGroupID++;
			}
			xmin = 0;
			ymin = ymin + ySize;
		}
		return Groups;
	}

}
```

A.5.1.2 SecondTestScenarioGroups

```
package overlay.server.core.groups;

import java.util.Vector;
import overlay.server.core.elements.Group;

public class SecondTestScenarioGroups implements GroupsCreator{

        /**
    * This method returns the Groups defined in the Second Test Scenario.
    *
        */
        public Vector<Group> getGroups() {
                Vector<Group> Groups = new Vector<Group>();
                Groups.add(new Group(1, 0, 9, 15, 20));
                Groups.add(new Group(2, 9, 13, 0, 15));
                Groups.add(new Group(3, 9, 20, 15, 20));
                return Groups;
        }

}
```

A.5.1.3 DefaultServerHandlerProcessor

```
package overlay.server.com.handlers.processor;

import overlay.server.core.GroupServer;
import overlay.server.core.elements.Group;
import overlay.server.core.elements.Node;
import overlay.server.util.GroupServerUtil;

public class DefaultServerHandlerProcessor implements ServerHandlerProcessor
    {

            public String addNode(String request, String IP){
                    Node newnode = GroupServerUtil.getNode(request, IP);
                         String response = "-ERR␣Node␣Null";
                         if(newnode!=null){
                                 int GroupID = processNewNode(newnode);
                                 if(GroupID!=0)
                                         response = GroupServerUtil.
                                             buildResponse(newnode, GroupID);
                                 else
                                         response = "-ERR␣Group␣Not␣Existant"
                                             ;
                         }
                         return response;
            }

            private int processNewNode(Node newnode){
                    int GroupID = 0;
```

```
                GroupID = GroupServer.dbase.addNode(newnode);
                return GroupID;
        }

        public String updateNode(String request, String IP){
                Node node = GroupServerUtil.getNodeWithID(request, IP);
                String response = "-ERR";
                if(node!=null){
                        int GroupID = processUpdateNode(node);
                        if(GroupID > 0)
                                response = GroupServerUtil.buildResponse(
                                    node, GroupID);
                }
                return response;
        }

        private int processUpdateNode(Node node){
                int GroupID = 0;
                GroupID = GroupServer.dbase.updateNode(node);
                return GroupID;
        }

        public String delNode(String request){
                String [] temp = request.split("&");
                if(temp.length<2)
                        return "-ERR Lack of Information";
                int NodeID = GroupServerUtil.getNodeID(temp[0]);
                int GroupID = GroupServerUtil.getGroupID(temp[1]);
                GroupServer.dbase.delNode(NodeID, GroupID);
                return "+OK NodeDeleted";
        }

        public String operation(String request){
                int GroupID = GroupServerUtil.getGroupID(request);
                Group updt = GroupServer.dbase.getGroupByID(GroupID);
                Node newCentralNode = updt.getLastUpdatedNode();
                updt.CentralNode = newCentralNode;
                GroupServer.dbase.setGroupByID(GroupID, updt);
                return "NodeID="+updt.CentralNode.NodeID;
        }
}
```

A.5.2 Overlay Nodes Classes

A.5.2.1 GetPositionFromSimulator

```
package overlay.core.position;

import overlay.com.ElementalHttpPost;
```

```
public class GetPositionFromSimulator implements GetPosition{

        private int x=-1;
        private int y=-1;
        private int ID=-1;
        private String host = "";
        private int port = 7000;

        public GetPositionFromSimulator(String host, int port, int ID){
                this.host = host;
                this.port = port;
                setID(ID);
        }

        public int getX(){
                return x;
        }

        public int getY(){
                return y;
        }

        public void setID(int ID) {
                this.ID = ID;
        }

        public void fetchXY() {
                ElementalHttpPost post = new ElementalHttpPost(host, port);
                String resp = "";
                this.x = -1;
                this.y = -1;
                try {
                    resp = post.Post("/fetch?"+"ID="+ID);
                    System.out.print(resp);
                    if(!resp.equals("-ERR_Data_Not_Found")){
                                String [] temp = resp.split("&");
                                for(int i=0; i<temp.length; i++){
                                        String [] temp2 = temp[i].split("=")
                                            ;
                                        if(temp2[0].equals("X"))
                                                this.x = Integer.parseInt(
                                                    temp2[1]);
                                        if(temp2[0].equals("Y"))
                                                this.y = Integer.parseInt(
                                                    temp2[1]);
                                }
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }

        }

}
```

A.5.2.2 OverlayUtil

```
package overlay.core.utils;

import java.io.IOException;
import java.net.InetAddress;
import java.net.UnknownHostException;

import overlay.com.handlers.processor.HandlerProcessor;
import overlay.com.servers.NodeHTTPServer;

import overlay.core.ClientOverlay;
import overlay.core.elements.Group;
import overlay.core.elements.Node;

public class OverlayUtil {

        /** Overlay Processor Utils **/

        /**
         * Gets this node data using the response from the Group Server.
         */
        public static Node getThisNode(String response, int x, int y){
                int NodeID = -1;
                String [] temp = response.split("\r\n");
                for(int i=0; i < temp.length; i++){
                        if(temp[i].equals("ThisNode")) {
                                String [] temp2 = temp[i+1].split("=");
                                NodeID = Integer.parseInt(temp2[1]);
                                break;
                        }
                }
                InetAddress addr;
                String IP = null;
                try {
                        addr = InetAddress.getLocalHost();
                        IP = addr.getHostAddress();
                } catch (UnknownHostException e) {
                        e.printStackTrace();
                }
                if(IP!=null)
                        return new Node(NodeID, x, y, IP);
                else
                        return new Node(NodeID, x, y, "localhost");
}

/**
 * Gets CentralNode data using the response from the Group Server;
 *
 * @param response Response from the Group Server;
 * @return Node CentralNode The notified central node;
 */
public static Node getCentralNode(String response){
        String [] temp = response.split("\r\n");
        int centralNodeID=-1, centralNodeX=-1, centralNodeY=-1;
        String centralNodeIP="";
        for(int i=0; i < temp.length; i++){
                if(temp[i].equals("CentralNode")) {
                        String [] temp2 = temp[i+1].split("=");
                        centralNodeID = Integer.parseInt(temp2[1]);
                        temp2 = temp[i+2].split("=");
                        centralNodeX =  Integer.parseInt(temp2[1]);
                        temp2 = temp[i+3].split("=");
                        centralNodeY =  Integer.parseInt(temp2[1]);
                        temp2 = temp[i+4].split("=");
                        centralNodeIP = temp2[1];
                        break;
                }
        }
        Node central = new Node(centralNodeID, centralNodeX,
            centralNodeY, centralNodeIP);
        central.Central();
        return central;
}

/**
* Gets GroupID using the response from the Group server.
*
* @param response Response from the Group server.
* @return int groupID The ID of the node's group.
*/
public static int getGroupID(String response){
        String [] temp = response.split("\r\n");
        int groupID = -1;
        for(int i=0; i < temp.length; i++){
                String [] temp2 = temp[i].split("=");
                if(temp2[0].equals("GroupID")) {
                        groupID = Integer.parseInt(temp2[1]);
                        break;
                }
        }
        return groupID;
}
```

```
/**
 * Builds the group using the response from the main server.
 *
 * @param response Response from the main server;
 */
public static void addGroupNodes(String response){
        String [] temp = response.split("Nodes\r\n");
        String [] temp1 = temp[1].split("\r\n");
        for(int i=1; i < temp1.length; i++){
                String [] temp2 = temp1[i].split("␣");
                int nodeid = Integer.parseInt(temp2[0]);
                if(nodeid != ClientOverlay.Group.ThisNode.NodeID){
                        int x = Integer.parseInt(temp2[1]);
                        int y = Integer.parseInt(temp2[2]);
                        ClientOverlay.Group.addNode(new Node(nodeid,
                             x, y, temp2[3]));
                }
        }
}

/**
 * Iniciates the Node's HTTP Server to process requests.
 *
 * @throws IOException
 */
public static void NodeHTTPServer(HandlerProcessor pc){
        try {
                new NodeHTTPServer(ClientOverlay.Group.ThisNode.
                    NodeID, pc);
        } catch (IOException e) {
                e.printStackTrace();
        }
}

/**
 * Creates a new empty Group.
 *
 * @param GroupID The ID of the Group;
 */
public static void defineGroup(int GroupID){
        ClientOverlay.Group = new Group(GroupID);
}

/**
 * Gives this node's IP.
 *
 * @return String This node's IP;
 */
```

```
public static String getThisNodeIP(){
        String IP = ClientOverlay.Group.ThisNode.IP;
        try {
                InetAddress addr = InetAddress.getLocalHost();
                IP = addr.getHostAddress();
        } catch (UnknownHostException e) {
                e.printStackTrace();
        }
        return IP;
}

/** Handler Processor Utils **/

/**
* Gets a node object from a received query.
*/
public static Node getNode(String input, String IP, boolean aux){
        String [] temp = input.split("&");
        int NodeID = -1;
        int x = -1;
        int y = -1;
        String ip = null;
        for(int i=0; i<temp.length; i++){
                String [] temp2 = temp[i].split("=");
                if(temp2[0].equals("NodeID"))
                        NodeID = Integer.parseInt(temp2[1]);
                if(temp2[0].equals("X"))
                        x = Integer.parseInt(temp2[1]);
                if(temp2[0].equals("Y"))
                        y = Integer.parseInt(temp2[1]);
                if(temp2[0].equals("IP"))
                        ip = temp2[1];
        }
        if(ip==null || aux){
                String [] newIP = IP.split(":");
                return new Node(NodeID, x, y, newIP[0]);
        }
        else
                return new Node(NodeID, x, y, ip);
}

/**
* Gets a node object from the database that has the same NodeID as
    in the received query.
* @param String input - Received query;
* @return Node with NodeID equal to the one in the received query;
*/
public static Node getNode(String input){
    String [] temp = input.split("=");
```

```
            if(temp[0].equals("NodeID")){
                    int nodeid = Integer.parseInt(temp[1]);
                    if(ClientOverlay.Group.ThisNode.NodeID == nodeid)
                            return ClientOverlay.Group.ThisNode;
                    else
                            return ClientOverlay.Group.getNodeByID(nodeid);
            }
            else
                    return null;
        }

        /**
        * Method that parses a node's query and gets the integer value where
             the parameter equals the type String.
        */
        public static int getData(String input, String type){
                String [] temp = input.split("&");
                for(int i=0; i<temp.length; i++){
                        String [] temp2 = temp[i].split("=");
                        if(temp2[0].equals(type)){
                                return Integer.parseInt(temp2[1]);
                        }
                }
                return -2;
        }

        /**
        * This method does the same as the one before but it returns a
            String instead of an integer.
        */
        public static String getDataString(String input, String type){
                String [] temp = input.split("&");
                for(int i=0; i<temp.length; i++){
                        String [] temp2 = temp[i].split("=");
                        if(temp2[0].equals(type))
                                return temp2[1];
                }
                return null;
        }
}
```

A.5.2.3 DefaultOverlayProcessor

```
package overlay.core.processor;

import overlay.com.handlers.processor.DefaultHandlerProcessor;
import overlay.com.requests.GroupServerRequests;
import overlay.com.requests.OverlayRequests;
import overlay.core.ClientOverlay;
import overlay.core.elements.Node;
import overlay.core.utils.OverlayMessagesUtil;
import overlay.core.utils.OverlayUtil;

public class DefaultOverlayProcessor implements OverlayProcessor{

        public int NodeID;
        public int X;
        public int Y;

        public DefaultOverlayProcessor(){
        }

        public void changeXY(int x, int y){
                if(ClientOverlay.Group != null)
                        NodeID = ClientOverlay.Group.ThisNode.NodeID;
                X = x;
                Y = y;
        }

        public int processResponseAddNode(String response){
                int groupID = -1;
                groupID = OverlayUtil.getGroupID(response);
                if(groupID < 0)
                        return -1;
                else
                        OverlayUtil.defineGroup(groupID);
                ClientOverlay.Group.defineThisNode(OverlayUtil.getThisNode(
                    response, X, Y));
                ClientOverlay.Group.addNode(OverlayUtil.getCentralNode(
                    response));
                OverlayUtil.addGroupNodes(response);
                NodeJoin(false);
                return groupID;
        }

        /**
        * Method to initiate the HTTP server and send a join request to all
            the nodes in the group.
        *
        * @param Boolean hasServer it is true if the client is already
            running an HTTP server;
        */
        private void NodeJoin(boolean hasServer){
                System.out.println(">>␣Joining␣Group");
                if(!hasServer)
                        OverlayUtil.NodeHTTPServer(new
                            DefaultHandlerProcessor());
                if(ClientOverlay.Group.CentralNode.NodeID != ClientOverlay.
                    Group.ThisNode.NodeID)
                System.out.println(OverlayRequests.join(
                    ClientOverlay.Group.CentralNode, ClientOverlay.
                    Group.ThisNode, ""));
        System.out.println("===============");
}

public String groupLeaveNode() {
        String resp = "-ERR Group Leave";
        if(ClientOverlay.Group.ThisNode.NodeID == ClientOverlay.
            Group.CentralNode.NodeID){
                if(ClientOverlay.Group.GroupNodes.size()>0){
                        System.out.println(">> Getting New Central 
                            Node");
                        resp = GroupServerRequests.Operation();
                        System.out.println(">> Updating Nodes");
                        centralNodeLeave(resp);
                        System.out.println("===============");
                }
                else
                        resp = "+OK Group Terminated";
        }
        else
                resp = OverlayRequests.leave(ClientOverlay.Group.
                    CentralNode, ClientOverlay.Group.ThisNode, "");
        System.out.println(resp);
        return resp;
}


/**
* Method to select a new central node inform all the nodes in the
    group of the new central node and that the actual central node is
     leaving;
*
* @param Node New central node;
* @return String with response from other nodes;
*/
private String centralNodeLeave(String newnode){
        String [] node = newnode.split("=");
        int NodeID = -1;
        if(node[0].equals("NodeID"))
                NodeID = Integer.parseInt(node[1]);
        String resp = "-ERR Central Node Leave";
        if(NodeID > 0)
                for(int i=0; i<ClientOverlay.Group.GroupNodes.size()
                    ; i++){
                    resp = OverlayRequests.leave(ClientOverlay.Group.
                        GroupNodes.get(i), ClientOverlay.Group.
                        ThisNode, "&NewCentralNodeID="+NodeID);
                }
        return resp;
}

public int processResponseUpdateNode(String response){
        int oldgroupID = ClientOverlay.Group.GroupID;
        int groupID = -1;
        groupID = OverlayUtil.getGroupID(response);
        if(groupID < 0)
                return -1;
        else{
                if(oldgroupID != groupID){
                        groupLeaveNode();
                        OverlayUtil.defineGroup(groupID);
                        ClientOverlay.Group.defineThisNode(
                            OverlayUtil.getThisNode(response, X, Y));
                        ClientOverlay.Group.addNode(OverlayUtil.
                            getCentralNode(response));
                        OverlayUtil.addGroupNodes(response);
                        NodeJoin(true);
                }else{
                        if(ClientOverlay.Group.CentralNode.NodeID ==
                             ClientOverlay.Group.ThisNode.NodeID)
                                this.centralNodeUpdate();
                        else
                                this.normalNodeUpdate();
                }
        }
        return groupID;
}

/**
* Method to send update request to the central node in order to
    inform all the nodes in the group.
*/
private void normalNodeUpdate(){
        System.out.println(">>␣Updating␣Node␣in␣Group");
        String IP = OverlayUtil.getThisNodeIP();
        ClientOverlay.Group.ThisNode = new Node(NodeID, X, Y, IP);
        String resp = OverlayRequests.update(ClientOverlay.Group.
            CentralNode, ClientOverlay.Group.ThisNode, "");
        System.out.println(resp);
        System.out.println("================");
}

/**
* Method to send update requests to all the nodes in the group.
*/
private void centralNodeUpdate(){
        System.out.println(">>␣Updating␣Central␣Node␣in␣Group");
        String IP = OverlayUtil.getThisNodeIP();
        ClientOverlay.Group.ThisNode = new Node(NodeID, X, Y, IP);
        ClientOverlay.Group.CentralNode = ClientOverlay.Group.
            ThisNode;
        for(int i=0; i<ClientOverlay.Group.GroupNodes.size(); i++){
                String resp = OverlayRequests.update(ClientOverlay.
                    Group.GroupNodes.get(i), ClientOverlay.Group.
                    ThisNode, "");
                System.out.println(resp);
        }
        System.out.println("==============");
}

public String sendMsg(String message, int NodeID){
        message = OverlayMessagesUtil.codeMessage(message);
        String resp  = "-ERR␣Message␣Not␣Delivered";
        if(ClientOverlay.Group.ThisNode.NodeID == ClientOverlay.
            Group.CentralNode.NodeID)
                resp = OverlayRequests.sendMsg(ClientOverlay.Group.
                    getNodeByID(NodeID), ClientOverlay.Group.ThisNode
                    .NodeID, NodeID, message,"");
        else
                resp = OverlayRequests.sendMsg(ClientOverlay.Group.
                    CentralNode, ClientOverlay.Group.ThisNode.NodeID,
                     NodeID, message,"");
        return resp;
}

public String sendBroadcast(String message){
        String response = "+OK␣BroadcastDelivered";
        int notdelivered = 0;
        if(ClientOverlay.Group.ThisNode.NodeID == ClientOverlay.
            Group.CentralNode.NodeID){
                for(int i=0; i<ClientOverlay.Group.GroupNodes.size()
                    ; i++){
                   String resp = sendMsg(message, ClientOverlay.
                       Group.GroupNodes.get(i).NodeID);
                   if(resp.equals("-ERR␣MessageNotDelivered"))
                           notdelivered++;
                }
                response = "+OK␣BroadcastDelivered␣(-"+notdelivered+
                    ")";
        }
        else
                response = OverlayRequests.sendMsg(ClientOverlay.
                    Group.CentralNode, ClientOverlay.Group.ThisNode.
                    NodeID, -1, message,"");
        return response;
        }
}
```

A.5.2.4 DefaultHandlerProcessor

```
package overlay.com.handlers.processor;

import overlay.com.Logger;
import overlay.com.requests.OverlayRequests;

import overlay.core.ClientOverlay;
import overlay.core.elements.Node;
import overlay.core.utils.OverlayMessagesUtil;
import overlay.core.utils.OverlayUtil;

public class DefaultHandlerProcessor implements HandlerProcessor{

        public static int counter = 0;

        public DefaultHandlerProcessor(){

        }

        /**
        * Method to send a message to the central node with the node
            information
        * with the type being join or update;
        */
        private void presentGroupNode(Node node, String type){
                for(int i=0; i<ClientOverlay.Group.GroupNodes.size(); i++){
                        Node temp = ClientOverlay.Group.GroupNodes.elementAt
                            (i);
                        if(temp.NodeID!=node.NodeID & temp.NodeID !=
                            ClientOverlay.Group.CentralNode.NodeID)
                          System.out.println(OverlayRequests.transmitNode(
                              node, temp, type, ""));
                }
        }

        public String joinGroup(String input, String IP){
                boolean CentralNode = false;
                if(ClientOverlay.Group.ThisNode.NodeID == ClientOverlay.
                    Group.CentralNode.NodeID)
                        CentralNode = true;
                Node newnode = OverlayUtil.getNode(input, IP, CentralNode);
                if(newnode == null)
                        return "-ERR Lack of Information";
                if(CentralNode)
                        presentGroupNode(newnode, "join");
                ClientOverlay.Group.addNode(newnode);
        return "+OK␣Node␣Joined";
}

public String updateGroup(String input, String IP){
        boolean CentralNode = false;
        if(ClientOverlay.Group.ThisNode.NodeID == ClientOverlay.
            Group.CentralNode.NodeID)
           CentralNode = true;
        Node node = OverlayUtil.getNode(input, IP, CentralNode);
        if(node == null)
                return "-ERR␣Lack␣of␣Information";
        if(CentralNode)
                presentGroupNode(node, "update");
        updateGroupNode(node);
        return "+OK␣Node␣Updated";
}

private void updateGroupNode(Node node){
        Node oldnode = ClientOverlay.Group.getNodeByID(node.NodeID);
        if(oldnode != null){
                ClientOverlay.Group.GroupNodes.remove(oldnode);
                ClientOverlay.Group.GroupNodes.add(node);
        }
}

public String leaveGroup(String input){
        Node nodeleave = ClientOverlay.Group.getNodeByID(OverlayUtil
            .getData(input, "NodeID"));
        if(nodeleave.NodeID == ClientOverlay.Group.CentralNode.
           NodeID){
                Node newcentral = ClientOverlay.Group.getNodeByID(
                    OverlayUtil.getData(input, "NewCentralNodeID"));
                newcentral.Central();
                if(nodeleave == null || newcentral == null){
                        System.out.println("ERRO!");
                        return "-ERR␣Node␣does␣not␣exist";
                }
                delGroupNode(nodeleave);
                ClientOverlay.Group.addNode(newcentral);
        }
        else{
                if(nodeleave == null){
                        System.out.println("ERRO!");
                        return "-ERR␣Node␣does␣not␣exist";
                }
                if(ClientOverlay.Group.ThisNode.NodeID ==
                    ClientOverlay.Group.CentralNode.NodeID)
                        presentGroupNode(nodeleave, "leave");
                delGroupNode(nodeleave);
        }
        return "+OK␣Node␣Leaved";
    }

    private String delGroupNode(Node nodeleave){
        String resp = "+OK␣Node␣Leaved";
        if(nodeleave.NodeID == ClientOverlay.Group.CentralNode.
            NodeID){
                resp = "+OK␣Central␣Node␣Leaved";
        }
        ClientOverlay.Group.delNode(nodeleave);
        System.out.println(resp);
        return resp;
    }

    public String messageHandler(String request){
        String resp = "-ERR␣MessageNotDelivered";
        int ThisNodeID = ClientOverlay.Group.ThisNode.NodeID;
        int DestNodeID = OverlayUtil.getData(request, "TO");
        int SenderNodeID = OverlayUtil.getData(request, "FROM");
        String message = OverlayUtil.getDataString(request, "MSG");
        if(DestNodeID<-1){
                resp = "-ERR␣MessageNotDelivered";
                return resp;
        }
        if(DestNodeID==-1){
                if(ThisNodeID == ClientOverlay.Group.CentralNode.
                    NodeID)
                        resp = sendBroadcast(SenderNodeID, message);
                else
                        resp = "+OK␣MessageDelivered";
                message = OverlayMessagesUtil.decodeMessage(message)
                    ;
                System.out.println(">>␣Received␣Broadcast␣Message:␣"
                    +message);
        }
        else
                if(DestNodeID == ThisNodeID){
                        message = OverlayMessagesUtil.decodeMessage(
                            message);
                        System.out.println(">>␣Received␣Message:␣"+
                            message);
                        resp = "+OK␣MessageDelivered";
                        counter++;
                        if(counter==1000){
                                Logger log = new Logger("localhost",
                                    9000);
                                log.Log(message, false);
                                counter = 0;
                                }
                        }
                        else
                                if(ThisNodeID == ClientOverlay.Group.
                                    CentralNode.NodeID)
                                        resp = sendMessage(SenderNodeID,
                                            DestNodeID, message);
                return resp;
        }

        private String sendMessage(int SenderID, int DestID, String message)
            {
                Node Dest = ClientOverlay.Group.getNodeByID(DestID);
                return OverlayRequests.sendMsg(Dest, SenderID, DestID,
                    message, "");
        }

        private String sendBroadcast(int SenderID, String message){
                int notdelivered = 0;
                String resp = "-ERR␣MessageNotDelivered";
                for(int i=0; i<ClientOverlay.Group.GroupNodes.size(); i++){
                        Node temp = ClientOverlay.Group.GroupNodes.get(i);
                        if(temp.NodeID!=SenderID){
                                resp = sendMessage(SenderID, temp.NodeID,
                                    message);
                                if(!resp.equals("+OK␣MessageDelivered"))
                                        notdelivered++;
                        }
                }
                resp = "+OK␣BroadcastDelivered␣(-"+notdelivered+")";
                return resp;
        }
}
```

A.5.2.5 TestOverlayUtil

```
package overlay.core.utils;

import java.util.Vector;

import overlay.core.ClientOverlay;
import overlay.core.elements.Node;

public class TestOverlayUtil {

        private static Vector<Node> getNewGroupNode(int NodeID){
                Vector <Node> temp = new Vector<Node>();
                for(int i=0; i<ClientOverlay.Group.GroupNodes.size(); i++){
                        temp.add(ClientOverlay.Group.GroupNodes.get(i));
                }
```

```
            for(int i=0; i<temp.size(); i++){
                    Node tempnode = temp.get(i);
                    if(tempnode.NodeID == NodeID)
                            temp.remove(tempnode);
            }
            temp.add(ClientOverlay.Group.ThisNode);
            return temp;
    }

    public static Vector<Node> getNeighbourNodes(){
            if(ClientOverlay.Group.GroupNodes.size()<3)
                    return ClientOverlay.Group.GroupNodes;
            Vector<Node> NeighbourNodes = new Vector<Node>();
            int NodeInd1 = 0;
            int Dif1 = Math.abs(ClientOverlay.Group.GroupNodes.get(0).
                NodeID-ClientOverlay.Group.ThisNode.NodeID);
            int NodeInd2 = 1;
            int Dif2 = Math.abs(ClientOverlay.Group.GroupNodes.get(1).
                NodeID-ClientOverlay.Group.ThisNode.NodeID);
            for(int i=2; i<ClientOverlay.Group.GroupNodes.size(); i++){
                    int Dif = Math.abs(ClientOverlay.Group.GroupNodes.
                        get(i).NodeID - ClientOverlay.Group.ThisNode.
                        NodeID);
                    if(Dif1 > Dif2){
                            if(Dif < Dif1){
                                    Dif1 = Dif;
                                    NodeInd1 = i;
                            }
                    }
                    else{
                            if(Dif < Dif2){
                                    Dif2 = Dif;
                                    NodeInd2 = i;
                            }
                    }
            }
            NeighbourNodes.add(ClientOverlay.Group.GroupNodes.get(
                NodeInd1));
            NeighbourNodes.add(ClientOverlay.Group.GroupNodes.get(
                NodeInd2));
            return NeighbourNodes;
    }

    public static Vector<Node> getNeighbourNodes(Node node){
            Vector<Node> NewGroupNodes = getNewGroupNode(node.NodeID);
            if(NewGroupNodes.size()<3)
                    return NewGroupNodes;
            Vector<Node> NeighbourNodes = new Vector<Node>();
            int NodeInd1 = 0;
```

```
                int Dif1 = Math.abs(NewGroupNodes.get(0).NodeID-node.NodeID)
                    ;
                int NodeInd2 = 1;
                int Dif2 = Math.abs(NewGroupNodes.get(1).NodeID-node.NodeID)
                    ;
                for(int i=2; i<NewGroupNodes.size(); i++){
                        int Dif = Math.abs(NewGroupNodes.get(i).NodeID -
                            node.NodeID);
                        if(Dif1 > Dif2){
                                if(Dif < Dif1){
                                        Dif1 = Dif;
                                        NodeInd1 = i;
                                }
                        }
                        else{
                                if(Dif < Dif2){
                                        Dif2 = Dif;
                                        NodeInd2 = i;
                                }
                        }
                }
                NeighbourNodes.add(NewGroupNodes.get(NodeInd1));
                NeighbourNodes.add(NewGroupNodes.get(NodeInd2));
                return NeighbourNodes;
        }
}
```

A.5.2.6 TestOverlayProcessor

```
package overlay.core.processor;

import java.util.Vector;

import overlay.com.handlers.processor.TestHandlerProcessor;
import overlay.com.requests.OverlayRequests;
import overlay.core.ClientOverlay;
import overlay.core.elements.Node;
import overlay.core.utils.OverlayMessagesUtil;
import overlay.core.utils.OverlayUtil;
import overlay.core.utils.TestOverlayUtil;

public class TestOverlayProcessor implements OverlayProcessor{

        private int NodeID;
        private int X;
        private int Y;
        private int TTL;
        private Vector<Node> NeighbourNodes;

        /**
```

```
* Constructor of the TestOverlayProcessor Class.
*/
public TestOverlayProcessor(){
        defineParameters();
}

/**
* Define the TTL and this node's neighbours.
*/
private void defineParameters(){
        if(ClientOverlay.Group != null){
                TTL = ClientOverlay.Group.GroupNodes.size();
                NeighbourNodes = TestOverlayUtil.getNeighbourNodes()
                    ;
        }
}

public void changeXY(int x, int y){
        if(ClientOverlay.Group != null)
                NodeID = ClientOverlay.Group.ThisNode.NodeID;
        X = x;
        Y = y;
}

public int processResponseAddNode(String response){
        int groupID = -1;
        groupID = OverlayUtil.getGroupID(response);
        if(groupID < 0)
                return -1;
        else
                OverlayUtil.defineGroup(groupID);
        ClientOverlay.Group.defineThisNode(OverlayUtil.getThisNode(
            response, X, Y));
        OverlayUtil.addGroupNodes(response);
        defineParameters();
        NodeJoin(false);
        return groupID;
}

/**
* Method to initiate the HTTP server and send a join request to the
    neighbour nodes in the group.
*
* @param Boolean hasServer it is true if the client is already
    running an HTTP server;
*/
private void NodeJoin(boolean hasServer){
        System.out.println(">> Joining Group");
        String Headers = "&TTL="+TTL+"&SenderID="+ClientOverlay.
            Group.ThisNode.NodeID;
        if(!hasServer)
                OverlayUtil.NodeHTTPServer(new TestHandlerProcessor
                    ());
        for(int i=0; i<NeighbourNodes.size(); i++){
                String resp = OverlayRequests.join(NeighbourNodes.
                    get(i), ClientOverlay.Group.ThisNode, Headers);
                System.out.println(resp);
        }
        System.out.println("===============");
}

public int processResponseUpdateNode(String response){
        int oldgroupID = ClientOverlay.Group.GroupID;
        int groupID = -1;
        groupID = OverlayUtil.getGroupID(response);
        defineParameters();
        if(groupID < 0)
                return -1;
        else{
                if(oldgroupID != groupID){
                        System.out.println(groupLeaveNode());
                        OverlayUtil.defineGroup(groupID);
                        ClientOverlay.Group.defineThisNode(
                            OverlayUtil.getThisNode(response, X, Y));
                        OverlayUtil.addGroupNodes(response);
                        NodeJoin(true);
                }else
                        NodeUpdate();
        }
        return groupID;
}

/**
* Method to send update request to the neighbour nodes in order to
    inform all the nodes in the group.
*/
private void NodeUpdate(){
        System.out.println(">>␣Updating␣Node␣in␣Group");
        String Headers = "&TTL="+TTL+"&SenderID="+ClientOverlay.
            Group.ThisNode.NodeID;
        String IP = OverlayUtil.getThisNodeIP();
        ClientOverlay.Group.ThisNode = new Node(NodeID, X, Y, IP);
        for(int i=0; i<NeighbourNodes.size(); i++){
                String resp = OverlayRequests.update(NeighbourNodes.
                    get(i), ClientOverlay.Group.ThisNode, Headers);
                System.out.println(resp);
        }
```

```
            System.out.println("===============");
        }

        public String groupLeaveNode() {
            String resp = "-ERR Group Leave";
            String Headers = "&TTL="+TTL+"&SenderID="+ClientOverlay.
                Group.ThisNode.NodeID;
            for(int i=0; i<NeighbourNodes.size(); i++){
                resp = OverlayRequests.leave(NeighbourNodes.get(i),
                    ClientOverlay.Group.ThisNode, Headers);
                System.out.println(resp);
            }
            return resp;
        }

        public String sendMsg(String message, int NodeID){
            message = OverlayMessagesUtil.codeMessage(message);
            String Headers = "&TTL="+TTL+"&MID="+OverlayMessagesUtil.
                getMessageID();
            Vector <Node> ClosestNodes = TestOverlayUtil.
                getNeighbourNodes();
            for(int i=0; i<ClosestNodes.size(); i++){
                    OverlayRequests.sendMsg(ClosestNodes.get(i),
                        ClientOverlay.Group.ThisNode.NodeID, NodeID,
                        message, Headers);
            }
            return "+OK MessageDelivered";
        }

        public String sendBroadcast(String message){
            message = OverlayMessagesUtil.codeMessage(message);
            String Headers = "&TTL="+TTL+"&MID="+OverlayMessagesUtil.
                getMessageID();
            Vector <Node> ClosestNodes = TestOverlayUtil.
                getNeighbourNodes();
            for(int i=0; i<ClosestNodes.size(); i++){
                    OverlayRequests.sendMsg(ClosestNodes.get(i),
                        ClientOverlay.Group.ThisNode.NodeID, -1, message,
                         Headers);
            }
            return "+OK BroadcastDelivered";
        }
}
```

A.5.2.7 TestHandlerProcessor

```
package overlay.com.handlers.processor;

import java.util.Vector;
import overlay.com.Logger;
import overlay.com.requests.OverlayRequests;

import overlay.core.ClientOverlay;
import overlay.core.elements.Node;
import overlay.core.utils.OverlayMessagesUtil;
import overlay.core.utils.OverlayUtil;
import overlay.core.utils.TestOverlayUtil;

public class TestHandlerProcessor implements HandlerProcessor{

        public static int counter = 0;
        public int TTL = 0;
        public int SenderNodeID = -2;
        public static long [] MessageID = new long[5000];

        public TestHandlerProcessor(){
        }

        /**
        * Method to send a request to the neighbour nodes with the node
            information
        * with the type being join, update or leave;
        */
        private void presentGroupNode(Node node, String type){
                String Headers = "&TTL="+TTL+ "&SenderID=" + ClientOverlay.
                    Group.ThisNode.NodeID;
                Vector<Node> CloserNodes = TestOverlayUtil.getNeighbourNodes
                    ();
                Vector<Node> NodeClosest = TestOverlayUtil.getNeighbourNodes
                    (node);
                if(ClientOverlay.Group.GroupNodes.size()>2)
                        for(int i=0; i<CloserNodes.size(); i++){
                                Node temp = CloserNodes.get(i);
                                if(temp.NodeID!=node.NodeID && temp.NodeID!=
                                    SenderNodeID
                                                && temp.NodeID!=NodeClosest.
                                                    get(0).NodeID && temp.
                                                    NodeID!=NodeClosest.get
                                                    (1).NodeID){
                                        OverlayRequests.transmitNode(node,
                                            temp, type, Headers);
                                }
                        }
        }

        public String joinGroup(String input, String IP){
                boolean CentralNode = false;
                SenderNodeID = OverlayUtil.getData(input, "SenderID");
            Node newnode = OverlayUtil.getNode(input, IP, CentralNode);
            if(newnode == null || ClientOverlay.Group.getNodeByID(
                newnode.NodeID) != null || SenderNodeID < 0)
                    return "-ERR Lack of Information";
            ClientOverlay.Group.addNode(newnode);
            TTL = OverlayUtil.getData(input, "TTL") - 1;
            if(TTL>0 && ClientOverlay.Group.ThisNode.NodeID!=
                ClientOverlay.Group.getMinNodeID()
                        && ClientOverlay.Group.ThisNode.NodeID!=
                            ClientOverlay.Group.getMaxNodeID()){
                    presentGroupNode(newnode, "join");
            }
            return "+OK Node Joined";
    }

    public String updateGroup(String input, String IP){
            boolean CentralNode = false;
            SenderNodeID = OverlayUtil.getData(input, "SenderID");
            Node node = OverlayUtil.getNode(input, IP, CentralNode);
            if(node == null || SenderNodeID < 0)
                    return "-ERR Lack of Information";
            updateGroupNode(node);
            TTL = OverlayUtil.getData(input, "TTL") - 1;
            if(TTL>0 && ClientOverlay.Group.ThisNode.NodeID!=
                ClientOverlay.Group.getMinNodeID()
                        && ClientOverlay.Group.ThisNode.NodeID!=
                            ClientOverlay.Group.getMaxNodeID()){
                    presentGroupNode(node, "update");
            }
            return "+OK Node Updated";
    }

    private void updateGroupNode(Node node){
            Node oldnode = ClientOverlay.Group.getNodeByID(node.NodeID);
            if(oldnode != null){
                    ClientOverlay.Group.GroupNodes.remove(oldnode);
                    ClientOverlay.Group.GroupNodes.add(node);
            }
    }

    public String leaveGroup(String input){
            Node nodeleave = ClientOverlay.Group.getNodeByID(OverlayUtil
                .getData(input, "NodeID"));
            SenderNodeID = OverlayUtil.getData(input, "SenderID");
            System.out.println("Node Trying to Leave: "+ ClientOverlay.
                Group.getNodeByID(OverlayUtil.getData(input, "NodeID")));
            if(nodeleave == null || SenderNodeID<0){
                    System.out.println("ERRO!");
                    return "-ERR Node does not exist";
```

```
          }
          TTL = OverlayUtil.getData(input, "TTL") - 1;
          if(TTL>0 && ClientOverlay.Group.ThisNode.NodeID!=
              ClientOverlay.Group.getMinNodeID()
                         && ClientOverlay.Group.ThisNode.NodeID!=
                              ClientOverlay.Group.getMaxNodeID()){
                    presentGroupNode(nodeleave, "leave");
          }
          ClientOverlay.Group.delNode(nodeleave);
          return "+OK␣Node␣Leaved";
}

public String messageHandler(String request){
          String resp = "-ERR␣MessageNotDelivered";
          int ThisNodeID = ClientOverlay.Group.ThisNode.NodeID;
          int DestNodeID = OverlayUtil.getData(request, "TO");
          int SenderNodeID = OverlayUtil.getData(request, "FROM");
          String message = OverlayUtil.getDataString(request, "MSG");
          TTL = OverlayUtil.getData(request, "TTL") - 1;
          long NewMessageID = Long.parseLong(OverlayUtil.getDataString
              (request, "MID"));
          if(MessageID[SenderNodeID] == NewMessageID){
                    resp = "-ERR␣MessageAlreadyDelivered";
                    return resp;
          }
          if(DestNodeID<-1){
                    resp = "-ERR␣MessageNotDelivered";
                    return resp;
          }
          MessageID[SenderNodeID] = NewMessageID;
          if(DestNodeID==-1){
                    if(TTL > 0)
                              resp = sendBroadcast(SenderNodeID, message,
                                   TTL);
                    else
                              resp = "+OK␣MessageDelivered";
                    System.out.println(">>Received␣Broadcast␣Message:␣"+
                         OverlayMessagesUtil.decodeMessage(message));
                    counter++;
                    if(counter==1000){
                              Logger log = new Logger("localhost", 9000);
                              log.Log(message, false);
                       counter = 0;
                    }

          }
          else
                    if(DestNodeID == ThisNodeID){
                        System.out.println(">>Received_Message:_"+
                            OverlayMessagesUtil.decodeMessage(message
                            ));
                        resp = "+OK_MessageDelivered";
                }
                else
                        if(TTL > 0)
                                resp = propagateMessage(SenderNodeID
                                    , DestNodeID, message, TTL);
        return resp;
}

private String sendBroadcast(int SenderID, String message, int TTL){
        String Headers = "&TTL="+TTL+"&MID="+MessageID[SenderID];
        if(ClientOverlay.Group.GroupNodes.size()>2){
                Vector<Node> NodeClosest = TestOverlayUtil.
                    getNeighbourNodes(ClientOverlay.Group.getNodeByID
                    (SenderID));
                Vector<Node> ClosestNodes = TestOverlayUtil.
                    getNeighbourNodes();
                for(int i=0; i<ClosestNodes.size(); i++){
                        Node temp = ClosestNodes.get(i);
                        if(temp.NodeID!=SenderID && temp.NodeID!=
                            NodeClosest.get(0).NodeID && temp.NodeID
                            !=NodeClosest.get(1).NodeID)
                                System.out.println(OverlayRequests.
                                    sendMsg(temp, SenderID, -1,
                                    message, Headers));
                }
        }
        return "+OK_BroadcastDelivered";
}

private String propagateMessage(int SenderID, int DestID, String
    message, int TTL){
        String Headers = "&TTL="+TTL+"&MID="+MessageID[SenderID];
        Node Dest = ClientOverlay.Group.getNodeByID(DestID);
        Vector<Node> ClosestNodes = TestOverlayUtil.
            getNeighbourNodes();
        String resp = "-ERR_Message_Not_Delivered";
        if(ClosestNodes.contains(Dest))
                return OverlayRequests.sendMsg(Dest, SenderID,
                    DestID, message, Headers);
        else{
                Vector<Node> NodeClosest = TestOverlayUtil.
                    getNeighbourNodes(ClientOverlay.Group.getNodeByID
                    (SenderID));
                for(int i=0; i<ClosestNodes.size(); i++){
                        Node temp = ClosestNodes.get(i);
```

```
                                if(temp.NodeID!=SenderID && temp.NodeID!=
                                    NodeClosest.get(0).NodeID && temp.NodeID
                                    !=NodeClosest.get(1).NodeID)
                                        resp = OverlayRequests.sendMsg(temp,
                                             SenderID, DestID, message,
                                            Headers);
                        }
                }
                return resp;
        }
}
```

A.5.3 Simulator Classes

A.5.3.1 ServerDatabaseTextFile

```
package overlay.simulator.core;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class SimulatorDatabaseTextFile implements SimulatorDatabase{

        private boolean end = false;
        private int [] x = new int[1000];
        private int [] y = new int[1000];
        private BufferedReader input;

        /**
        * Construtor of the Database Class.
        *
        * @param String With the text filename to be read;
        */
        public SimulatorDatabaseTextFile(String file){
                File f = new File(file);
                try {
                        input = new BufferedReader(new FileReader(f));
                } catch (FileNotFoundException e) {
                        e.printStackTrace();
                }
        }

        /**
        * Method to get the next line from a text file.
        *
        * @return String The next line from the text file;
        */
```

```
    private String getNextLine(){
            String line = null;
            try {
                    line = input.readLine();
            } catch (IOException e) {
                    e.printStackTrace();
            }
            return line;
    }

    /**
    * Gets the next position given by X and Y from the text file.
    */
    public void fetchXY(){
            String line = getNextLine();
            if(line != null){
                    System.out.println(">>␣Processing␣line:");
                    System.out.println(line.replace("\t", "␣|␣"));
                    String [] temp = line.split("\t");
                    for(int i=0; i<temp.length; i++){
                            String [] temp2 = temp[i].split("&");
                            int NodeID = -1;
                            for(int j=0; j<temp2.length; j++){
                                    String [] temp3 = temp2[j].split("="
                                        );
                                    if(temp3[0].equals("ID")){
                                            NodeID = Integer.parseInt(
                                                temp3[1]);
                                            if(NodeID!=-1){
                                                    x[NodeID] = -1;
                                                    y[NodeID] = -1;
                                            }
                                    }
                                    if(temp3[0].equals("X") && NodeID !=
                                        -1)
                                            x[NodeID] = Integer.parseInt
                                                (temp3[1]);
                                    if(temp3[0].equals("Y") && NodeID !=
                                        -1)
                                            y[NodeID] = Integer.parseInt
                                                (temp3[1]);
                            }
                    }
                    System.out.println("==============");
            }
            else
                    end = true;
    }
```

```
/**
* Gives back the actual value of Y of a Node.
*
* @param int ID − Identification of the node;
* @return int Y − Actual value of Y;
*/
public int getX(int NodeID){
        if(end)
                return −1;
        else
                return x[NodeID];
}

/**
* Gives back the actual value of Y of a Node.
*
* @param int ID − Identification of the node;
* @return int Y − Actual value of Y;
*/
public int getY(int NodeID){
        if(end)
                return −1;
        else
                return y[NodeID];
}

/**
* Function to see if there is more position values.
*
* @return true  If there is not more position values to fecth;
* @return false Otherwise;
*/
public boolean isEnd(){
        return end;
}

/**
* Function to force the end boolean to be true in order to end the
    simulation.
*/
public void setEnd(){
        end = true;
}

/**
* Build a String to return via an HTTP response.
*
* @return String for HTTP response.
*/
```

```
        public String buildResponse(String input) {
                int NodeID = getID(input);
                String resp = "-ERR Data Not Found";
                if(NodeID > 0)
                        resp = "X="+getX(NodeID)+"&Y="+getY(NodeID);
                return resp;
        }

        /**
        * Method to get the ID from a node's request in order to return its
            position.
        *
        * @param String Request from the node;
        * @return int The ID from the request node's position;
        */
        private int getID(String input){
                String [] temp = input.split("=");
                int NodeID = -1;
                if(temp[0].equals("ID"))
                   NodeID = Integer.parseInt(temp[1]);
           return NodeID;
        }

}
```

References

[1] Lockheed Martin Federal Systems Odetics Intelligent Transportation Systems Division. Its executive summaries. December 1999.

[2] E. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Survey and Tutorial*, March 2004.

[3] Pedro Garcia, Carles Pairot, Ruben Mondejar, Jordi Pujol, Helio Tejedor, and Robert Rallo. *PlanetSim: A New Overlay Network Simulation Framework.* 2005.

[4] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. OverSim: A Flexible Overlay Network Simulation Framework. In *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA*, pages 79–84, May 2007.

[5] G. Finn and J. Touch. Network construction and routing in geographic overlays. *ISI Technical Report ISI-TR-2002-564*, July 2002.

[6] J. Navas and T. Imielinski. Geocast - geographic addressing and routing. *Proceedings of the 3rd annual ACM/IEEE international conference on Mobile computing and networking*, September.

[7] T. Imielinski and J. Navas. Gps-based addressing and routing. *RFC2009*, March 1996.

[8] Das, S. Bowles, B. Houghland, C. Hunn, and S. Zhang. Microscopic simulations of freeway traffic flow. *Simulation Symposium, 1999. Proceedings. 32nd Annual*, pages 79–84, July-Sept. 1999.

[9] O. Andrisano, R. Verdone, and M. Nakagawa. Intelligent transportation systems: The role of third-generation mobile radio networks. *Communications Magazine, IEEE*, 38(9):144–151, September 2000.

[10] D. Ni. Determining traffic-flow characteristics by definition for application in its. *Intelligent Transportation Systems, IEEE Transactions on*, 8(2):181–187, June 2007.

[11] W. Collier and R. Weiland. Smart cars, smart highways. *Spectrum, IEEE*, 31(4):27–33, April 1994.

[12] K. Shenai, E. McShane, and M. Trivedi. Electronics technologies for intelligent transportation systems. *Intelligent Transportation System, 1997. ITSC 97. IEEE Conference on*, pages 302–307, November 1997.

[13] V. Venkatasubramanian and H. Leung. A robust chaos radar for collision detection and vehicular ranging in intelligent transportation systems. *Intelligent Transportation Systems, 2004. Proceedings. The 7th International IEEE Conference on*, pages 548–552, October 2004.

[14] Y. Zhao. Mobile phone location determination and its impact on intelligent transportation systems. *Intelligent Transportation Systems, IEEE Transactions on*, 1(1):55–64, March 2000.

[15] U.S. Department of Transportation. The national its architecture version 6.0. [online] http://itsarch.iteris.com/itsarch/index.htm.

[16] European Commission. The karen european its framework architecture. [online] http://www.frameonline.net/.

[17] R. Meier, A. Harrington, and V. Cahill. A framework for integrating existing and novel intelligent transportation systems. *Intelligent Transportation Systems, 2005. Proceedings. 2005 IEEE*, pages 154–159, September 2005.

[18] K. Biesecker, E. Foreman, K. Jones, and B. Staples. Intelligent transportation systems (its) information security analysis. November 1997.

[19] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *ACM SIGOPS Operating Systems Review*, 39(5):75–90, December 2005.

[20] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[21] Y. Zhao. *Decentralized Object Location and Routing: A New Networking Paradigm.* PhD thesis.

[22] J. Berkes. Decentralized peer-to-peer network architecture: Gnutella and freenet. 2003.

[23] The Gnutella v0.4 protocol. [online] http://www9.limewire.com/developer/gnutella_protocol_ 0.4.pdf.

[24] Gnucleus The gnutella web caching system. [online] http://www.gnucleus.com/gwebcache/.

[25] I. Clarke, O. Sandberg, B. Wiley, and H. Theodore. *Freenet: A Distributed Anonymous Information Storage and Retrieval System.* 2001.

[26] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. *Towards a common API for structured peer-to-peer overlays*, pages 33–44. Number Volume 2735/2003 in Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2003.

[27] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.

[28] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[29] Y. Zhao, L. Huang, J. Stribling, C. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *Selected Areas in Communications, IEEE Journal on*, 22(1):41–53, 2004.

[30] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[31] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[32] X-BONE Overlay System. [online] http://www.isi.edu/xbone/.

[33] P2: Declarative Network. [online] http://p2.cs.berkeley.edu/p2related.php.

[34] W. Burghout. *Hybrid microscopic-mesoscopic traffic simulation.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2004.

[35] Microsimulation of road traffic. [online] http://vwisb7.vkw.tu-dresden.de/ treiber/microapplet/.

[36] FreeSim: A Free Real-Time Freeway Traffic Simulator. [online] http://www.freewaysimulator.com/.

[37] Simetron Metropolitan Traffic Simulator. [online] http://www.olympum.com/ bruno/simetron.html.

[38] A. Varga. Omnet++ community site. [online]. available: http://www.omnetpp.org/.

[39] Eclipse IDE. [online] http://www.eclipse.org/.