Software Infrastructure for Communication in Distributed Robotics Systems

Ali Osman ISIK

Master of Science Thesis

TU Delft Delft University of Technology

Delft Center for Systems and Control

Software Infrastructure for Communication in Distributed Robotics Systems

Master of Science Thesis

For the degree of Master of Science in Embedded Systems at Delft University of Technology

Ali Osman ISIK

November 25, 2010

Supervisors

Prof.dr. R. Babuska

Dr. G.A.D. Lopes

A. Simonetto

Faculty of Electrical Eng., Mathematics and Computer Science · Delft University of Technology

Abstract

Distributed Control Systems (DCSs) are being more popular in process control, autonomous systems and safety-critical systems because of several advantages over centrally controlled systems. The Distributed Robotics Group at the Delft Center for Systems and Control (DCSC) is currently researching on distributed control of mobile networks. In order to to apply distributed control on a mobile network, agents in the network must communicate to each other.

Communication libraries that are created for Multi Agent Systems (MASs) and Distributed Robotics System (DRS) are generally application specific and do not address many systems with different capabilities. The purpose of this project is to design and implement a software communication library for DRSs not only to meet the needs of Distributed Robotics Group, but also to address more general MASs. The library is called the Distributed Robotics Library (DRL).

The DRL is modeled by the 7 Layer Open Systems Interconnection (OSI) Reference Model. Since Wi-Fi standards are used for the physical transmission of data between agents and the medium access control in the network, agents must be Wi-Fi certified or compliant in order to use this version of the library. For the network layer IPv4, for the transport control TCP and UDP protocols are used. These protocols provide all the necessary services for the DRL.

Our DRL is divided into two classes, one for general communication in the network and one for the specific communication with a robot. For identification of the robots in the network, an automatic advertisement broadcasting technique is used. For every identified robot in the network, an object of the specific communication class is created. Starting and stopping these specific communications are controlled by general communication class. This class is using connection request/accept technique to start server/client communication between the robots. All these processes in the library, are running in parallel in order to minimize the transmission delay and to maximize data transfer rate. However, resource management is not taken into consideration in this project.

The DRL is designed to detect the robots which leave the network or shut down for some reason. In these cases, failing robots are detected and removed from the network. The robots

which sleep or leave the network, automatically introduce themselves when they are back. However, they cannot recover previous information in this version of the library.

To test the DRL five Asus Eee PC's are used to represent the robots. I have successfully tested all the functions in the DRL library via a number of verification applications. During the tests, message relaying function is implemented and added as an extension to the library. Although all the PC's are running Microsoft Windows XP, the DRL was also tested a Unix environment to test its portability.

The current version of the DRL lays the ground towards the implementation of a powerful general purpose automatic communications library. In the DRL, the closest three layers to the user are implemented. By implementing lower layers, it can be made more efficient for distributed control applications.

Table of Contents

Acknowledgements

First I would like to thank my supervisor Prof.dr. R. Babuska for giving me the opportunity to do my MSc Thesis in DCSC and for his feedbacks on my final report. Special thanks to my daily supervisor Dr. G.A.D. Lopes for his tremendous contribution to the Distributed Robotics Library and for motivating me during the thesis. I also want to thank to A. Simonetto for giving significant feedbacks and providing the testbed that I used to test the Distributed Robotics Library. Last but not the least, I would like to thank my family and friends for their support.

Delft, University of Technology — Ali Osman ISIK
November 25, 2010

Chapter 1

Introduction

Distributed Control Systems (DCSs) are a class of systems designed to control distributed processes or complete manufacturing systems by using decentralized elements or subsystems. In centralized control, a core controller is responsible for the (pre)planning, task-assignment and supervision of the task coordination. However in DCSs, agents are capable of sensing, acting, communicating and they contribute to the task solution all together. Unlike the centrally controlled systems where the central controller is fully interacted with the agents, DCSs are network centric systems and each agent requires only partial interaction with the other agents. Since the knowledge is distributed in the network, simpler architectures are enough for DCSs and agents have lower resource requirements than a central controller. However, each agent must have advanced communication attributes and agents should be interconnected by a communication network.

DCSs have significant advantages over centrally controlled systems. Since the whole system is not critically relying on any individual, the system is more robust and fault tolerant. In centralized systems, the knowledge is collected in one individual and the whole system is vulnerable against any attack on the knowledge in that individual. Distributed knowledge in DCSs makes the system less vulnerable against any attack that is trying to compromise the security of the system. Therefore, nowadays DCSs are being applied more often in process control, autonomous systems and safety-critical systems where control needs to change to cope with fault appearance or other process disturbance[1].

There are different DCS models have been proposed for different applications. For example, Galdun and Tamac [2] propose a DCS model consisting of 4 levels: technological level, supervisory level, information level and management level. By using supervisory level and information level where all the information is saved in huge databases, they try to obtain more reliable DCSs. Another model, which is proposed by Aguilar and Cerrada [3], is a model based on a hierarchical architecture to be used in automating industrial processes. In this hierarchy, each level consists of a group of different tasks to be carried out within the control system and each level is using and generating different information.

Studies on distributed control greatly benefit from the utilization of Multi Agent Systems (MASs), since they provide a platform for simulation and testing of various hypotheses based

on the principle of distributed artificial intelligence[4]. Thus, we can say that MASs are the new structure of decentralized control systems. Distributed control MAS approaches have been used in various domains, such as unmanned air, terrestrial/underwater vehicles, search and rescue scenarios, collective robotics, social cognition, etc [4]. In some of the applications control can also be distributed over agents and humans. For instance, Koes and Nourbakhsh [5] have studied hybrid rescue group systems based on humans, software agents, and autonomous robots. What they describe is a coordination architecture capable of quickly finding an optimal solution to the combined problems of task allocation, scheduling, and path planning subject to system constraints. However, Trianni and Dorigo [6] have focused on a distributed coordination which implies that the characteristics of the group's behaviour are not managed centrally by one or few "leaders" but are the result of self-organizing processes. In their project, groups of robots evolve a cooperative strategy for tasks which can be modeled as ant colonies. To summarize, DCSs use the coordination of agents to increase the efficiency of the control systems.

1-1 Focus and Contribution of the Thesis

The Distributed Robotics Group at the Delft Center for Systems and Control (DCSC) is planning to address many problems in distributed robotics area, especially about sensing and distributed control of mobile networks. In order to address these problems, a testbed which consists of multiple robots and sensors is being built as shown in Figure 1-1. Currently, there are 6 iRobots and each of them use a netbook for communication and running complex control algorithms. The testbed is surrounded by beacons for localization of the robots. Since each robot is able to perform complex calculations and can be programmed as an intelligent agent, the testbed is a MAS with intelligent agents and many distributed control applications can be tested.

In order to apply and test the distributed control techniques on this system, the agents must be interconnected by a communication network. Since each robot has netbooks and has wireless network adapters, the communication network can be set up by a wireless router. The main focus of this thesis is implementing a software infrastructure for communication of these netbooks, in other words, providing a software library to be used in a Distributed Robotics System (DRS). The library should be portable, running in both Microsoft Windows and Unix operating systems. By running parallel to the user applications, the library should automatically discover other robots in the network to provide simple communication for data transfer to be used by these applications. This kind of a communication library provides set of rules to create a common language for communication of different robots in a network.

1-2 Thesis Outline

In the first part of Chapter 2 we give background information about MASs, their modeling and communication aspects. The next part of Chapter 2 gives the background information about High Level Communication Protocols. We start with the 7 layer Open Systems Interconnection (OSI) model, which is the general model of a communication library, then we explain ZigBee, Wi-Fi and Bluetooth communication protocols. In Chapter 3, after introducing our

Figure 1-1: Distributed Robotics Lab.

Distributed Robotics Library (DRL), we explain the main functionalities of the library. Then, by the help of UML sequence diagram, we explain the execution of the library. In Chapter 4, we give a demonstration of the library on a testbed by setting up a communication network and exchanging internal data between netbooks. Chapter 5 gives the overview of the library and discusses some possible future work.

Chapter 2

Communication in Networks

The purpose of this chapter is to give a reference guide to networking. Most of the sections in this chapter serve as background information for the remainder of the thesis, and can be safely skipped for a less computer science inclined reader. However, they are important in presenting a general idea about networking and giving a better understanding about the design of the Distributed Robotics Library (DRL) that we developed. To maintain the fidelity to the original referenced document, some texts are cited verbatim. Since we are dealing with robotic networks in this thesis, our focus is on Multi Agent Systems (MASs) and their communication properties. In Section 2-1, we start by giving information about some properties of physical agents. Then, we explain how communication is modeled between agents with graph-theoretic approach. In Section 2-2, we explain the 7 layer Open Systems Interconnection (OSI) model which is the standard model for communication systems. In this section, we also give example protocols to the each layer. In Section 2-3, we explain Bluetooth, ZigBee, Wi-Fi, which is used in this version of the library, and Zero Configuration Networking protocols. Although the whole chapter is given as a reference guide to networking, Section 2-2 is the model that we used in our library and in the design chapter, we will use some of the concepts from this section. Also, Zero Configuration Networking subsection gives a strong method to build lower layers of a communication system for local networks and can a path for further improvements of our communication library.

2-1 Modeling of Multi Agent Systems

2-1-1 Physical Agents

Physical agents are situated in an environment of the real-world, they are embodied [7]. Since they are interacting with the physical environment, they have some specific individual characteristics to deal with the physical world. From a very abstract perspective, the basic architecture of a physical agent should consist of three components: a set of sensors, a set of effectors, and a cognitive capability. Although today's robots are missing cognitive part,

robots can be good examples of physical agents. We can list some of the main requirements for modeling a physical agent.

- A physical agent must have access to data in the physical world. Sensors are used for this purpose.
- In a physical distributed system, there should be a common global time and all agents should synchronize their clocks with that time.
- In a communication process of agents in a distributed environment, time representation should be integrated to the information transmitted.
- In a communication process of agents, there should be reliable protocols that assure the communication. If the application is real-time, these protocols must also assure the communication with low latency.
- There should be a controller that assures coherent use of resources and agents should implement hard resource management.
- Agents should be fault tolerant and should offer a recovery mechanism.
- If a physical agent is mobile, it should have adapting and learning mechanisms.

2-1-2 Geometric Model of Multi Agent Systems

MASs can be represented geometrically with the help of graphs in which agents correspond to nodes, and communication links correspond to edges.

Basics of Graph Theory

A simple graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of $\mathcal{V}$, a nonempty set of vertices and $\mathcal{E}$, a set of unordered pairs of distinct elements of $\mathcal{V}$ called edges. Simple graphs have their limits in modeling the real world. Instead, we use multi graphs, which consist of vertices and undirected edges between these vertices, with multiple edges between pairs of vertices allowed. Note that simple graphs are all multi graphs. In order to represent connection to itself pseudo graph is defined which allows loops in the edge set. Another type is directed graph $G = (\mathcal{V}, \mathcal{E})$ consists of a set $\mathcal{V}$ of vertices, a set $\mathcal{E}$ of edges, that are ordered pairs of elements of $\mathcal{V}$. The basic terminology that describes the vertices and edges of undirected graphs as follows [8]:

- Two vertices u and v in a graph $\mathcal{G}$ are called adjacent (or neighbors) in $\mathcal{G}$ if u, v is an edge of $\mathcal{G}$.
- The topological structure of a graph is represented by the adjacency matrix. Let N be the number of vertices. Then the adjacency matrix will be $N \times N$ where each element α_{ij} is either 1 if there is a connection or 0 if there is no edge connects vertices i and j.
- The degree of a vertex in a graph is the number of edges incident with it, except that a loop at a vertex contributes two to the degree of that vertex. The degree of the vertex v is denoted by $\deg(v)$ or $\mathrm{d}(v)$. The sum of degrees in a graph is double of the edges.

- A vertex of degree zero is called an isolated vertex.
- A complete graph has an edge between every pair of vertices.
- A weighted graph has weight on its edges which can be analogous to the real distance between corresponding vertices.

Connectivity

Many problems can be modeled with paths formed by traveling along the edges of graphs. For instance, the problem of determining whether a message can be sent between two computers using intermediate links can be studied with a graph model.

A "path" is a sequence of edges that begins at a source vertex and ends at a destination vertex in a graph. A graph is connected if there is a path between every pair of vertices. When there is no path for at least one source-destination pair, the graph is disconnected. Connectivity is an important measure for the robustness of a MAS and it makes easier to study and understand the properties of the system. A disconnected graph is union of two or more connected graphs. These disjoint subgraphs are called connected components of graphs. The largest of these components is called giant component. When ratio of giant component size and the graph size is a measure for connectivity. When it is closed to 1, the graph is closed to a connected graph and can be approximated to a connected graph to make the study easier.

In a graph, "hop count" specifies the number of hops on the path between a source and a destination. It is used to measure some concepts in graph theory such as "Small World Property". This is one of the most important phenomenon in connectivity of networks. A network is said to have the small-world property when the hop count in that network is not strongly affected by an increase in the network size. Many networks in real life show small world property. It also refers to the concept that everyone is connected to everyone else in the world by only six degrees of separation, or six sets of acquaintances.

"Clustering coefficient" of a node is the ratio between the actual number of links between the neighbors of node i and the maximum possible number of links between these neighbors. This coefficient is a measure to which nodes in a graph tend to cluster together. This leads us to the concept of "Local Correlation". Let node i be connected to node j. If the probability of node i being connected to the neighbors of node j is higher than the probability of node i being connected to other nodes in the network, we say that edges are locally correlated. Local correlation is a factor that increases the clustering coefficient.

Different levels of connectivity can be utilized to analyze systems. By definition, k-connectivity means the minimum clustering coefficient of a node is k. In other words, minimum number of links needed to be removed to get a disconnected graph is k. In most of the times, 1-connectivity is enough to analyze, however, if someone wants to set up more reliable networks, upper connectivity levels can be considered.

Basic Graph Models

For different MASs, different graph models can be the best suited for graphical representation. In this part, we are looking for the best model for wireless networks. Note that wireless

networks show small world property, has local correlation and randomly distributed over a network. The channels between nodes are RF channels and log-distance path loss model is assumed. In this model, RF signal level decreases with the distance so the coverage of a node is limited.

In Erdos and Renyi random graph model, We denote a random graph by $G_p(N)$, where N is the number of nodes in the graph and p is the probability of having a link (edge) between any two nodes. Very interesting property of this model is there is a critical probability at which a giant cluster forms. Above a certain value, a giant component emerges and spans almost entire network. In this manner, this graph model is very easy to analyze. When p is above that threshold, the graph is assumed to be connected. This model also shows small world property since any node can be connected with any other one and increase in number of nodes does not increase maximum hop count. In terms of showing small world property and random distribution of nodes, it can be used for wireless networks, however, it is not realistic since the link between any two nodes is random.

In order to take into account the way that radio waves propagate, pathloss geometric random graph model is presented. In this model a node is connected to another node if its in the range. The range is a simple circle around the node. Obviously, this model has a better solution for RF signals but it is not realistic since the range cannot be a circle. Also, it does not show small world property. The node and link distribution of these two models is given in Figure 2-1 [9]. As it is also mentioned on the figure, reality is somewhere in between.

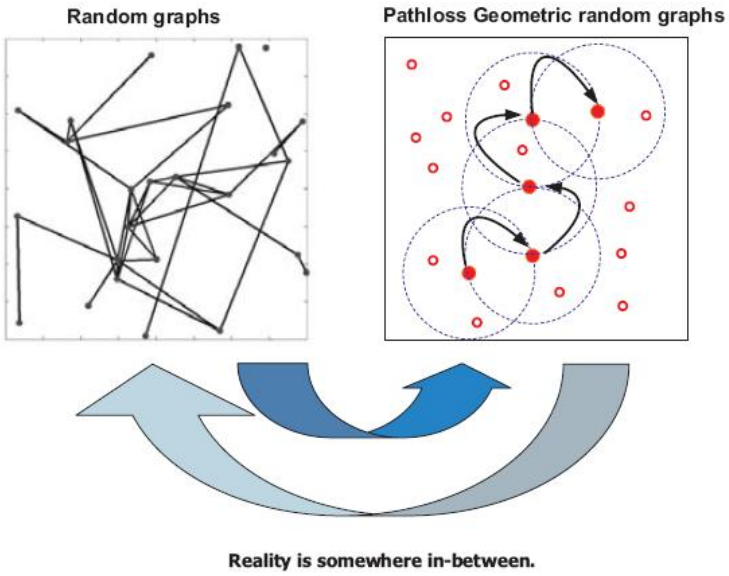

Figure 2-1: Comparison of Wireless Network Graph Models.

The most realistic model for wireless networks is lognormal geometric random graph model. In this model, log-distance path loss model is used for RF signals. The probability of a link between two nodes is a function the distance between them. This time, the range of a node is not represented by a circle but in a more realistic way. This model also shows small world property and has more realistic link distribution method, thus it is the best model for wireless networks.

In lognormal geometric random graph model, two ways can be used to check 1-connectivity:

1. Choosing a node at random and using a simple flooding algorithm to tag all nodes belonging to the same cluster. This procedure is repeated for all untagged nodes until no untagged nodes remain in the graph. If the largest cluster found in this way contains all nodes, the network is 1-connected.
2. The second procedure for checking 1-connectivity uses the $N \times N$ Laplacian of $\mathcal{G}$. By looking at the eigenvalues of laplacian, one can analyze many features about the connectivity.

In some cases, it is easier to determine connectivity by giant component size. If giant component includes all of the vertices, then the graph is 1-connected.

2-2 Modeling of Communication Libraries

2-2-1 7 Layer OSI Model

In a communication system, the transmitter and receiver must agree a set of rules between them before exchanging data. Protocols are those set of rules and they exist at many levels from physical connection to end-to-end services. The International Organization for Standardization (ISO) developed an abstract description for layered communications and network protocol design in 1974 which is called OSI Reference Model. This model can be applied to any communication system. It standardizes network architecture and encourages vendors to develop network equipments avoiding proprietary design. Basically, it divides network architecture into seven layers which, from top to bottom, from 7 to 1, are the Application, Presentation, Session, Transport, Network, Data-Link, and Physical Layers as shown in Figure 2-2. A layer is a collection of conceptually similar functions that provide services to the layer above it and receives service from the layer below it [10]. First three layers are the interface layers and the last four layers pertain to end-to-end functions such as user application, user services, user interface, session establishment. Two instances of the same layer are connected by a protocol connection on that layer. Therefore, in order to understand a communication library, first we should be aware of the properties of its layer.

Layer 1: Physical Layer

The Physical layer is the first and lowest layer in OSI model. The Physical Layer provides the mechanical, electrical, functional and procedural means to activate, maintain, and de-activate physical-connections for bit transmission between data-link-entities. A physical-connection may involve intermediate open systems, each relaying bit transmission within the physical Layer. The Physical Layer entities are interconnected by means of a physical medium [11]. As we stated before, the physical layer is supposed to provide some services for the data link layer. We can list some of the services of the physical layer as follows [10]:

- Providing physical interface specifications such as;

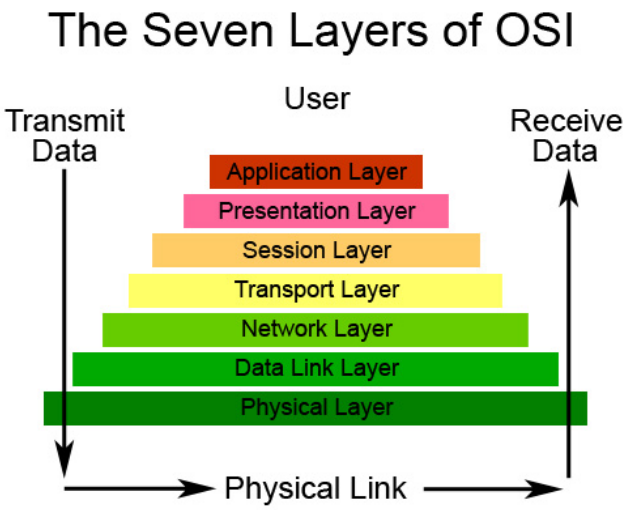

Figure 2-2: The 7 Layer OSI Model.

  - Mechanical specifications of wires and cables to ensure data delivery at the speed required.
  - Mechanical specifications of the connectors such as type, size, number of pins and pin configuration.
  - Electrical characteristics such as voltage level interpretation, current level maintanence, signaling type, duration, bandwidth and strength.

- Concerning with the transmission techniques such as synchronous or asynchronous, serial or parallel bit-by-bit or symbol-by-symbol transmission.
- Signaling start and stop of a transmission and controlling flow of the the transmission.
- Detecting fault conditions and providing them to the next level.
- Providing quality of service parameters such as error rate, service availability, transmission rate and transmission delay.

For different medium access protocols different physical layer specifications are presented. In these specifications, type of the wire, transceivers, transceiver cables, connectors, hub types, repeater types, terminators, distance limitations are varied. The most prominent physical layer protocols are RS232, USB, Ethernet Physical Layer, Token Ring Physical Layer, etc. These protocols are important but out of scope of this thesis. In Wireless Local Area Network (WLAN)(IEEE 802.11), five different transmission techniques can be used. Infrared, Frequency Hopping Spread Spectrum (FHSS), Direct Sequence Spread Spectrum (DSSS), Orthogonal Frequency Division Multiplexing (OFDM) and High Rate Direct Sequence Spread Spectrum (HR-DSSS) are used for physical data transmission.

Layer 2: Data Link Layer

The Data Link Layer (DLL) is the second level in OSI model. The DLL provides functional and procedural means for connectionless-mode among network-entities, and for connection-mode for the establishment, maintenance, and release data-link-connections among network-entities and for the transfer of data-link-service-data-units. A data-link-connection is built upon one or several physical-connections [11]. The DLL can be divided into two sublayers. The upper sublayer is the Logical Link Control (LLC) layer and the lower sublayer is the Media Access Control (MAC) layer.

Media Access Control Layer

The MAC sublayer acts as an interface between the physical layer and the LLC. This layer is not required for a full-duplex point-to-point communication but for the multi agent networks. We can list the services of this sublayer as follows:

- Unique addressing of each device in the network. This procedure is called MAC addressing. In Ethernet addressing 48 bits of addressing is used.
- It provides multiple access protocols for channel-access control. These protocols make it possible for several nodes to share the same medium. The most prominent protocols used for medium access will be explained in the following subsections.
- It is responsible for frame synchronization such as time based, character counting, byte stuffing and bit stuffing.
- It is responsible for switching, queuing or scheduling of data packets in the local area network.
- It controls quality of service by controlling the parameters provided by physical layer.
- When there is no physical network it establishes a Virtual Local Area Network (VLAN).

Now, we will look into two of most prominent multiple access control protocols in detail.

Carrier Sense Multiple Access with Collision Detection (CSMA/CD)

The first protocol solving the channel allocation problem was the ALOHA system. The idea behind this protocol is very simple. It allows the user the transmission whenever there is data to be sent. With a LAN a user can listen the channel while transmitting and can understand if there is a collision by a hardware feedback system. If listening while transmitting is not possible i.e. WLAN, a user can detect the collision by some acknowledgments. If the frame of a user is destroyed, the sender waits random amount of time and tries to transmit it again. These kind of systems are called contention systems since there is a contention between multiple users. In this kind of a system collision is of course inevitable. If we look at the efficiency of the ALOHA network, the best channel utilization we can get is 18 percent [12]. Since this result is not very encouraging a better method is presented as slotted ALOHA.

The only difference between ALOHA and slotted ALOHA is that time is divided into discrete intervals of one frame length. In this method, a transmission is only allowed at the beginning

of the time slots. In ALOHA network the vulnerable period was two frame size, however, in slotted ALOHA it is one frame size. So the efficiency is doubled and best channel utilization is 37 percent. This is still not very encouraging result.

In Local Area Networks (LANs), it is possible for users to detect what others are doing. So, to improve the performance Carrier Sense Multiple Access (CSMA) protocol is presented. In this protocol channels listen for carriers and act accordingly. One type is 1-persistent CSMA. In this protocol, when a station has something to send, it senses the channel. If channel is busy, it waits the channel to become idle. When it is idle, it transmits the frame. If a collision is detected, the station waits random amount of time and retransmits the frame. It gives a better channel utilization than slotted ALOHA. However, like ALOHA and slotted ALOHA, under heavy traffic, this method is also very inefficient due to high probability of collisions. A better solution under heavy traffic is non persistent CSMA. In this protocol, if a station finds the channel idle, it transmits. If the channel is busy it waits random amount of time and senses the channel again. Consequently, it gives a better channel utilization under heavy traffic but longer delays. Both persistent and non persistent protocols can detect the collision, but they still finish their transmissions.

To improve these protocols, a better protocol is presented called Carrier Sense Multiple Access With Collision Detection (CSMA/CD) (CSMA with collision detection). The improvement of this protocol is, whenever a channel senses a collision, it directly terminates the transmission, waits a random period of time and tries again. In Ethernet(IEEE 802.3), this method is used. In this protocol, the key issue is the time required to detect a collision in the worst case scenario. Note that the collision detection is an analog process and the station's hardware should be able to listen the cable while transmitting. Collision can be detected by comparing the power of received and transmitted signal. Worst case collision detection time can be found as follows:

- Assume that at time 0 station A starts transmission and let the propagation time between two furthest stations is τ.
- Assume that at time $\tau - \epsilon$ the furthest station B starts transmission unaware of the channel is busy.
- Then a collision occurs at time τ and station B detects the collision, aborts the transmission and sends noise bursts to let every station know about the collision.
- At time 2τ station A sees the noise burst and detects the collision.

So, worst case collision detection time is two times the maximum propagation delay in the network. This is the reason why there should be a minimum frame length in a network. If a station tries to transmit a short frame less that 2τ, then, the sender incorrectly concludes the transmission is successful before it detects the collision. Hence, if we notate the transmission time of a frame on the cable as "t", t should be larger than 2τ for collision detection. Note that τ increases with the distance of nodes in the network and decreases with the signal speed. t increases with the number of bits in a frame and decreases with transfer rate. Generally, the signal speed in the cable and transmission rate is fix and there is a trade-off between the other two parameters. Normally, we do not want the network size too small and minimum

frame size too high. However, with the increase of network size, minimum frame size should be increased and vice versa. In 10 mbps Ethernet(IEEE 802.3), worst case collision detection time is 51.2 μsec and network diameter is around 2500 meters. So minimum frame size is 512 bits which is 64 bytes.

Randomization of the waiting time after collision this algorithm works as follows:

- After the first collision, stations waits either 0 or 1 slot time. If each one picks the same number they will collide again.
- After the second collision, each station picks 0, 1, 2, 3 at random and waits the that much time slots.
- After ith collision, each station picks a random number between 0 and $2^i - 1$ and waits that much time slots.
- After 10th collision maximum interval is reached and it is 0-1023.
- After 16th collision the controller reports a failure and recovery is left to higher levels.
- One slot time is chosen as the worst case collision detection time which is 51.2 μsec in case of IEEE 802.3.

Because of some problems will be presented next part, CSMA/CD break down in wireless networks. So, for wireless networks, another CSMA method which is Carrier Sense Multiple Access With Collision Avoidance (CSMA/CA) is presented.

Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA)

The reason why CSMA/CD cannot be used in WLANs and WPAN is it is not possible listening while sending in these networks so collision detection is not possible. CSMA/CA is simply an improvement on CSMA because of two problem arises in wireless networks. In wireless networks, carrier sense range for a station is limited.

- Suppose that two stations which are not in the range of each other both trying to send data to a third station which is in the common range. Then, these two stations are hidden to each other so collision occurs. This is called *Hidden Node Problem.*
- Suppose that two stations which are in the range of each other both wants to send data to different stations out of mutual range. This should not cause any problem for the network. However, these two stations senses each others transmission and delay their transmissions since they find the channel busy. This is called *Exposed Node Problem.*

In CSMA/CA some improvements added to CSMA to solve these problems. It is supplemented by two signals Request to Send(RTS) and Clear to Send (CTS). In this protocol, the sender sends a RTS to the receiver which contains the length of the data. Then, receiver replies with CTS which includes the same information. Upon the receipt of CTS, the sender begins transmission. In this protocol, a station is not allowed to transmit data if it senses a CTS because it means that a station in the range is about to receive data. This solves Hidden

Node Problem. In contrast, a station which senses only RTS can start a transmission. As long as it does not collide with CTS, it can finish its transmission successfully and this property solves Exposed Node Problem. In spite of these improvements, a collision can still occur when two stations send RTS at the same time to a mutual receiver. CSMA/CA is used in WLAN 802.11. Unlike the CSMA/CD, CSMA/CA is also efficient under heavy traffic. There are some other CSMA methods for wireless LANs which can slightly change from CSMA/CA to improve the throughput.

Logical Link Control Layer

The LLC sublayer acts as an interface between the MAC sublayer and the Network Layer. This sublayer multiplexes the MAC protocols while transmitting and decoding while receiving. It encapsulates the Network Layer data packets into the frames. It also provides optional flow and error control in addition to the Transport Layer. In wireless networks this control is used with retransmission of the erroneous packets, however in LAN only error detection and canceling of the packet are provided. In most of the network protocols IEEE 802.2 standard is used as the LLC protocol. Now, we will explain this standard in detail.

IEEE 802.2

This standard provides a common interface for upper layers to deal with any kind of MAC protocols. It provides three types of operation for data communication.

- Type 1 is a connectionless service in which data is simply transmitted regardless of whether destination station accepted the packet. It allows multicast transmission.
- Type 2 is connection-oriented service. First a logical connection is established between two stations. The connection is maintained and terminated by using special packets that can be understood by stations. Sequence numbers are added to the packets for ordered transmission. Error control and flow control are provided but this service does not allow broadcast or multicast addresses.
- Type 3 is acknowledged connectionless service for point-to-point communication only.

The LLC header consists of 4 fields:

- 8-bit DSAP (Destination Service Access Point) is the SAP(Service Access Point) where the packet is intended.
- 8-bit SSAP (Source Service Access Point) is the SAP(Service Access Point) where the packet is originated.
- 8-bit control field for use in auxiliary functions such as flow control.
- Information field that carries LCC information.

IEEE 802.2 is independent of the particular medium and commonly used.

Layer 3: Network Layer

The Network Layer is the third level in OSI model. We can list the functions of this layer as follows:

- The Network Layer is responsible for the controlling the delivery of the packets from source to destination by attaching the packets its own control information. Remember that, the Data Link Layer is responsible for the node-to-node transmission of the frames which serves to the Network Layer.
- It divides messages into packets and resizes them to the sizing requirements of the receiving networks and it creates virtual circuit environments to deliver these packets in the correct sequence.
- It ensures the transmission speed is in the range that the receiver can candle.
- It maintains quality of service and error control functions.
- In connection-oriented networks, it establishes logical connection before the packet exchange.
- Responsible for the routing of packets from one network to the other network.
- It is responsible for unique host addressing of the nodes.

The best known protocol in the Network Layer is the Internet Protocol(IP). In the following subsection, we will briefly explain Internet Protocol.

Internet Protocol

The Internet Protocol has the task of delivering datagrams(pieces of data streams) from source to destination. The most used version of the Internet Protocol used in Internet is Internet Protocol Version 4(IPv4) and we will explain this version of Internet Protocol. In order to understand IP, first we should look into IPv4 datagram format. An IP datagram consists of a header and data part. The header has 20 byte fix part and optional 40 bytes. The rest of the datagram is the data part. In Figure 2-3 we see an IPv4 header with the name of the fields and definitions of the fields follow.

- **Version:** This field states which version of the protocol this datagram belongs to. In IPv4 case this field should be 4.
- **IHL:** Since the length of the header is not constant, IHL field tells the number of words in the header. Minimum value of this field is 5 and maximum is 15 which means header should be 20-60 bytes.
- **Type of Service:** The first three bits of this field are precedence field i.e. 111 : Network Control, 110: Internetwork control, 001: Priority, etc. Next three bits are D, T, R flags. If flag is set, it means corresponding (Delay, Throughput, Reliability) property is cared more.

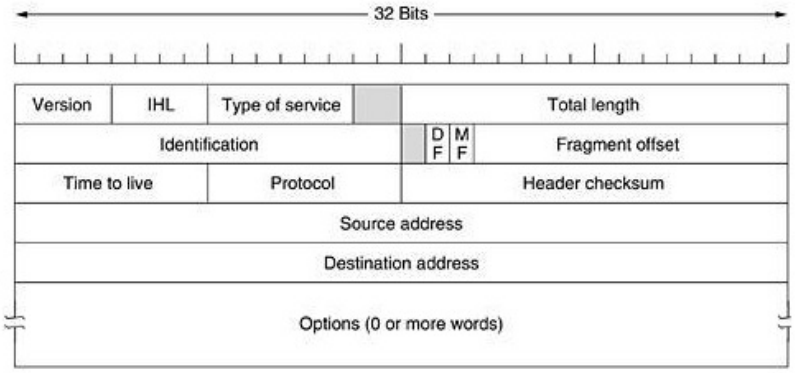

Figure 2-3: The IPv4 header.

- **Total length:** This field defines the datagram size. The maximum number is 65535 bytes and the minimum number is 20 bytes.
- **Identification:** It allows the destination host to determine which datagram an arrived fragment belongs to. The fragments of a datagram all contain the same identification value.
- **DF:** If this Don't Fragment(DF) bit is set, it means the destination host is incapable of defragmentation so it is an order not to fragment the datagram.
- **MF:** MF stands for More Fragment. All fragments except the last one have this bit set so it can be used to make sure all the fragments arrived.
- **Fragment Offset:** This field tells where this fragment belongs in the datagram in multiple of 8 bytes. In 13 bits, maximum fragments per datagram can be 8192. So, 65536 bytes can be represented which is enough when we consider the maximum length of 65535.
- **Time to live:** This field limits the life time of a datagram. It is a hop count field and it is decremented at every hop. When it hits 0, the packet is discarded and a warning packet is sent to the sender.
- **Protocol:** This field tells which protocol to be used at the transport layer after defragmentation of the fragments.
- **Header Checksum:** This field only verifies the header. At each hop header checksum is recomputed and compared with this field. When there is a mismatch the packet is discarded.
- **Source and Destination Addresses:** Every host on the Internet has a unique 32 bits IP address, which encodes its network number and host number. Until 1993, IP addresses were divided into 5 categories A, B, C, D, E which is called *classful addressing.* Classes A, B and C have different sizes for network field and host field which can all address small, medium and large networks. Class D is for multicast and class E is reserved for future use. Addresses are written in dotted decimal format as 4 octaves.

The lowest IP address is 0.0.0.0 and highest is 255.255.255.255. In this addressing all the hosts in a network have the same network number. However, with the increase in the network size, this IP addressing can cause some problems. Instead, in 1993 the network classes explained above were replaced by Classless Inter-Domain Routing (CIDR) which allows redivision of classes. There are a lot of issues in IP addressing which will not be discussed here.

- **Options:** Options field can be up to 40 bytes. Some of the IP options are as follows:
  - Security option tells the level of secrecy of the datagram.
  - Strict source routing option gives a complete path from source to destination.
  - Loose source routing gives a list of routers not to be missed.
  - Record route allows each router to append its IP address.
  - Timestamp option allows each router to append its IP address and timestamp.

In IP, at fragmentation stage, total length, MF, fragment offset and header checksum fields are computed and a header is put to all fragments of the IP datagram. In reassembly stage, fragments are assembled by looking at the values in their headers and datagram is passed to the next level for further usage.

Layer 4: Transport Layer

The Transport Layer is the fourth level in OSI model. It is the middle layer of the OSI model and Transport Layer data stream is a logical connection between end points. The upper layers of the OSI model are concerned with application issues and the lower levels are concerned with transmission issues [10]. We can list the services provided by the Transport Layer as follows:

- Segmentation of upper level services and reassembly of lower level packets. By segmentation, it encapsulates application data blocks to data segments suitable for transmission on the Network Layer.
- When the Network Layer does not guarantee the delivery of packets in the correct order, Transport Layer provides it by giving numbers to each packet.
- It guarantees the reliable transmission of the data from origin to destination. It establishes a high level of error checking and it recovers errors by retransmission of erroneous packets.
- Maintenance of information flow control when it is not provided by the Network Layer.

The Transport Layer protocols can be classified into five groups

- **Class 0 (TP0) :** This is the simplest protocol for connection-oriented networks. It performs no error checking or flow control.

- **Class 1 (TP1) :** It can monitor for transmission errors and recover the error by resending it. So, it provides reliable service.
- **Class 2 (TP2) :** It can monitor for transmission errors and provides flow control between Transport Layer and Session Layer.
- **Class 3 (TP3) :** In addition to the functions of TP1 and TP2 it has more reliable recovery mechanism.
- **Class 4 (TP4) :** In addition to the functions of TP3 it can be used for connectionless networks and it has more extensive error monitoring and recovery. It is the type closest to the TCP protocol.

In the following subsections we will explain two protocols used by Internet: User Datagram Protocol(UDP) and .

User Datagram Protocol (UDP)

User Datagram Protocol (UDP) is a connectionless protocol which provides a way for applications to send encapsulated IP datagrams and send them without a connection [12]. UDP does not provide guarantee for message delivery so it is also called Unreliable Datagram Protocol. However, it provides application multiplexing via port numbers and integrity verification via checksum. UDP transmits segments consisting of 8 byte header followed by the payload. UDP header consists 4 fields.

- The source port is needed when a reply must be sent back to the source. When it is not used it is assumed to be 0.
- The destination port is required and identifies the destination port.
- 16-bit length field identifies the length of the datagram in bytes.
- Checksum is optional and it is not used in IPv4 while it is used in IPv6.

UDP is a very simple protocol and it does not do flow control, error control or retransmission of a bad segment. All it does is, providing an interface to IP protocol with demultiplexing processes using ports. If an application requires reliability, it should use Transmission Control Protocol (TCP) protocol. UDP is especially useful in client-server situations. Since it generally assumes there is no error or correction is not necessary, it is preferred in time-sensitive applications where transmission delay is not acceptable. The common applications of UDP are Domain Name System (DNS), streaming media applications such as Voice over IP (VoIP), Trivial File Transfer Protocol (TFTP).

Transmission Control Protocol(TCP)

In this subsection, some texts are cited verbatim from the book [13] in order to maintain fidelity to the original referenced document. TCP provides a connection oriented, reliable, byte stream service. The term connection-oriented means the two applications using TCP must establish a TCP connection with each other before they can exchange data. TCP provides following facilities:

- **Stream Data Transfer:** TCP transfers a contiguous stream of bytes by grouping the bytes in TCP segments and passing them to IP for transmission to the destination.
- **Reliability:** TCP assigns a sequence number to each byte transmitted, and expects a positive acknowledgment(ACK) from the receiving TCP. If the ACK is not received within a timeout interval, the data is retransmitted. The receiving TCP uses the sequence numbers to rearrange the segments when they arrive out of order, and to eliminate duplicate segments.
- **Flow Control:** To allow for many processes within a single host to use TCP communication facilities simultaneously, the TCP provides a set of addresses or ports within each host. Concatenated with the network and host addresses from the internet communication layer, this forms a socket. A pair of sockets uniquely identifies each connection.
- **Logical Connections:** The reliability and flow control mechanisms described above require that TCP initializes and maintains certain status information for each data stream. The combination of this status, including sockets, sequence numbers and window sizes, is called a logical connection. Each connection is uniquely identified by the pair of sockets used by the sending and receiving processes.
- **Full Duplex:** Each TCP connection supports a pair of byte streams, one flowing in each direction.

TCP Header

TCP data is encapsulated in an IP datagram. The Figure shows the format of the TCP header. Its normal size is 20 bytes unless options are present. Each of the fields is discussed below:

- The **SrcPort** and **DstPort** fields identify the source and destination ports, respectively.
- The **sequence number** identifies the byte in the stream of data from the sending TCP to the receiving TCP that the first byte of data in this segment represents.
- The **Acknowledgement number** field contains the next sequence number that the sender of the acknowledgment expects to receive. This is therefore the sequence number plus 1 of the last successfully received byte of data. This field is valid only if the ACK flag is on. Once a connection is established the Ack flag is always on.
- The **Acknowledgement**, **SequenceNum**, and **AdvertisedWindow** fields are all involved in TCP's sliding window algorithm. The Acknowledgement and AdvertisedWindow fields carry information about the flow of data going in the other direction. In TCP's sliding window algorithm the receiver advertises a window size to the sender. This is done using the AdvertisedWindow field. The sender is then limited to having no more than a value of AdvertisedWindow bytes of un acknowledged data at any given time. The receiver sets a suitable value for the AdvertisedWindow based on the amount of memory allocated to the connection for the purpose of buffering data.
- The **header length** gives the length of the header in 32-bit words.

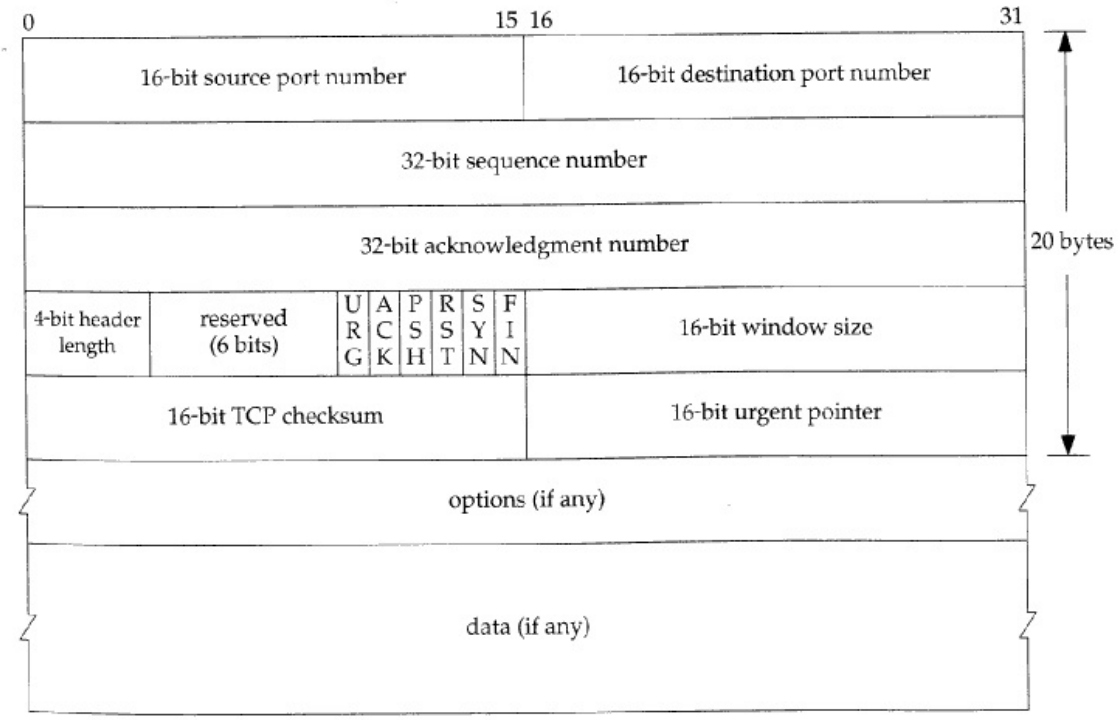

Figure 2-4: TCP Header.

- The **6-bit Flags** field is used to relay control information between TCP peers. The possible flags include SYN, FIN, RESET, PUSH, URG, and ACK.
  - The **SYN** and **Fin** flags are used when establishing and terminating a TCP connection, respectively.
  - The **ACK** flag is set any time the Acknowledgement field is valid, implying that the receiver should pay attention to it.
  - The **URG** flag signifies that this segment contains urgent data. When this flag is set, the **UrgPtr** field indicates where the non-urgent data contained in this segment begins.
  - The **PUSH** flag signifies that the sender invoked the push operation, which indicates to the receiving side of TCP that it should notify the receiving process of this fact.
  - The **RESET** flag signifies that the receiver has become confused and so wants to abort the connection.
- The **Checksum** covers the TCP segment: the TCP header and the TCP data. This is a mandatory field that must be calculated by the sender, and then verified by the receiver.
- The **Option** field is the maximum segment size option, called the MSS. Each end of the connection normally specifies this option on the first segment exchanged. It specifies the maximum sized segment the sender wants to receive.

- The **data** portion of the TCP segment is optional.

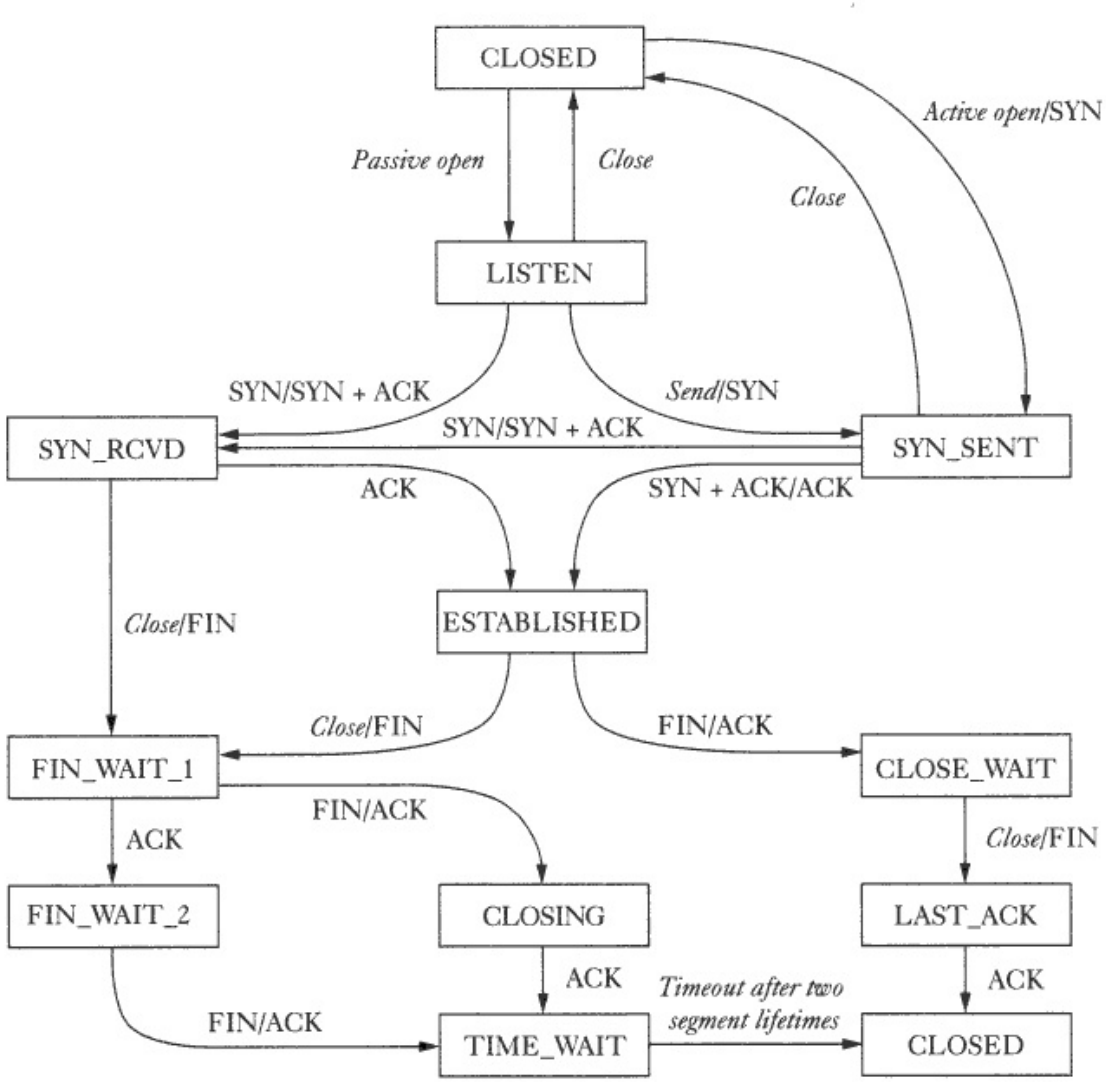

Figure 2-5: TCP State Transition Diagram.

TCP State Transition Diagram

The two transitions leading to the ESTABLISHED state correspond to the opening of a connection, and the two transitions leading from the ESTABLISHED state are for the termination of a connection. The ESTABLISHED state is where data transfer can occur between the two ends in both the directions.

If a connection is in the LISTEN state and a SYN segment arrives, the connection makes a transition to the SYN_RCVD state and takes the action of replying with an ACK+SYN segment. The client does an active open which causes its end of the connection to send a SYN segment to the server and to move to the SYN_SENT state. The arrival of the SYN+ACK segment causes the client to move to the ESTABLISHED state and to send an ACK back to

the server. When this ACK arrives the server finally moves to the ESTABLISHED state. In other words, we have just traced the THREE-WAY HANDSHAKE.

In the process of terminating a connection, the important thing to keep in mind is that the application process on both sides of the connection must independently close its half of the connection.

The main thing to recognize about connection teardown is that a connection in the TIME_WAIT state cannot move to the CLOSED state until it has waited for two times the maximum amount of time an IP datagram might live in the internet. The reason for this is that while the local side of the connection has sent an ACK in response to the other side's FIN segment, it does not know that the ACK was successfully delivered. As a consequence this other side might re-transmit its FIN segment, and this second FIN segment might be delayed in the network. If the connection were allowed to move directly to the CLOSED state, then another pair of application processes might come along and open the same connection, and the delayed FIN segment from the earlier incarnation of the connection would immediately initiate the termination of the later incarnation of that connection.

Sliding Window Algorithm

TCP implements an algorithm for flow control called **Sliding Window**. The *window* is the maximum amount of data we can send without having to wait for ACKs. In summary, the operation of the algorithm is as follows:

1. Transmit all the new segments in the window.
2. Wait for acknowledgment/s to come.
3. Slide the window to the indicated position and set the window size to the value advertised in the acknowledgment

When we wait for an acknowledgment to a packet for some time and it has not arrived yet, the packet is retransmitted. When the acknowledgment arrives, it causes the window to be repositioned and the transmission continues from the packet following the one transmitted last.

The sliding window serves the following purposes:

- It guarantees the reliable delivery of data.
- It ensures that the data is delivered in order.
- It enforces flow control between the sender and the receiver.

Layer 5: Session Layer

The Session Layer is the fifth layer in OSI model. The purpose of the Session Layer is to provide the means necessary for cooperating presentation entities to organize and to synchronize their dialogue and to manage their data exchange [11]. The services of the Session Layer in a connection mode transmission are as follows:

- It establishes session connection between two presentation entities upon requests of both entities and releases connection upon request of one of the entities.
- It provides normal data transfer and expedited data transfer between two presentation entities.
- It provides session connection synchronization which allows presentation entities to define synchronization points and reset the session connection.
- It notifies the presentation entities about the exceptions.
- In connectionless mode, the Session Layer only provides connectionless mode transmission and exception reporting.

In many protocols Session Layer is not given as an explicit layer like in TCP/IP suite. In TCP/IP there is only Application Layer above Transport Layer.

Layer 6: Presentation Layer

The Presentation Layer is the sixth layer in OSI model. This layer is responsible for formatting of and the delivery of information to the Application Layer for further process. We can list some of the services of the Presentation Layer as follows:

- Providing common representation of data to be transferred between application entities. That relieves application entities form concerning any problem of common representation of information and gives application entities syntax freedom.
- Encryption can also be done in this layer.
- It is responsible for selecting transfer syntax between application entities.

In many applications, there is no distinction between the Application Layer and the Presentation Layer. Most of the Application Layer protocols have Presentation Layer aspects, too. American Standard Code for Information Interchange (ASCII) is the most prominent Presentation Layer standard.

Layer 7: Application Layer

The Application Layer is the last layer in OSI model and closest to the end user. This is the OSI layer that interacts with the software applications implementing communication components. The services provided by this layer can be listed as follows:

- Identifying the communication partners and determining the data to transmit.
- Determining whether sufficient network resources available for the communication.
- Deciding whether the requested communication exists.
- It synchronizes the communication.

In this layer there are a lot of protocols. The most prominent ones are HyperText Transfer Protocol(HTTP), File Transfer Protocol (FTP), Domain Name System Protocol(DNS), etc.

2-2-2 Interface

OSI Reference Model does not specify any programming interfaces, other than abstract service specifications. The protocol specifications given in the OSI model define the interfaces between communicating agents, however there can be implementation specific software interfaces in each agent. For instance, Microsoft Windows' Winsock, and Unix's Berkeley sockets and System V Transport Layer Interface, are interfaces between Layer 5 and Layer 4.

2-3 High-level Communication Protocols

2-3-1 Bluetooth Protocol

In 1994, Erickson Mobile Communications became interested in low-cost low-consumption radio interface between mobile phones and other devices. In 1998 IBM, Intel, Nokia and Toshiba joined the project. They formed a SIG(Special Interest Group) to develop a short-range, low-power, low-cost wireless radios for communication devices and the project was named "Bluetooth". The first specifications was published in 1999 by Bluetooth SIG and it was the basis of IEEE 802.15 standards which is about Wireless Personal Area Networks(WPAN).

Architecture

Communication in bluetooth is based on master-slave principle. A master and up to seven slaves within a distance of 10 meters form a cell called piconet. Several piconets can overlap and form a scatternet as we see in Figure 2-6 [14]. Master node is responsible for channel allocation, communication establishment, setting the piconet synchronization clock and deciding for the frequency hopping sequence(FHS). Slaves cannot talk directly each other except discovery phase. A slave can be part of different piconets and a master can be slave of another piconet, as well.

In Bluetooth communication layering structure is slightly different than OSI model. IEEE 802.15 version of Bluetooth protocol architecture is given in Figure 2-7 [12]. Now, we look into these layers in detail.

Bluetooth Radio Layer

Since this is a physical layer, it moves bits between master to slave. The radio technology used is called frequency-hopping spread spectrum with 1600 hops/sec. In this method channels are defined in spread spectrum and every transmission is done in different frequency(channel). Bluetooth operates at 2.4 GHz ISM band and except France and Spain, this band is divided into 79 channels of 1 MHz each. In Spain and France it is divided into 23 channels of 1 Mhz each. Until Version 2.0 Gaussian frequency-shift keying (GFSK) is used for frequency modulation and with 1 bit per hertz 1 Mbps data rate can be reached. In 2.0 and later versions different methods are used and 2 or 3 Mbps data rates are reached.

There are 3 different types of devices in Bluetooth.

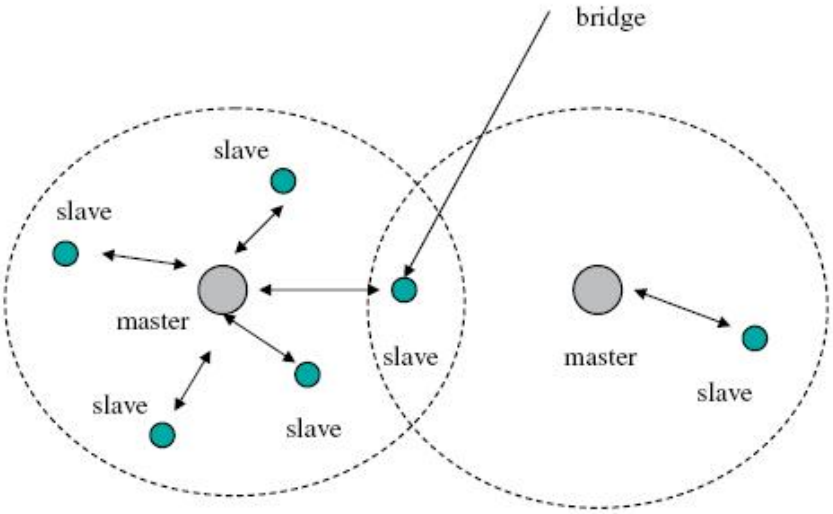

Figure 2-6: A Bluetooth Scatternet.

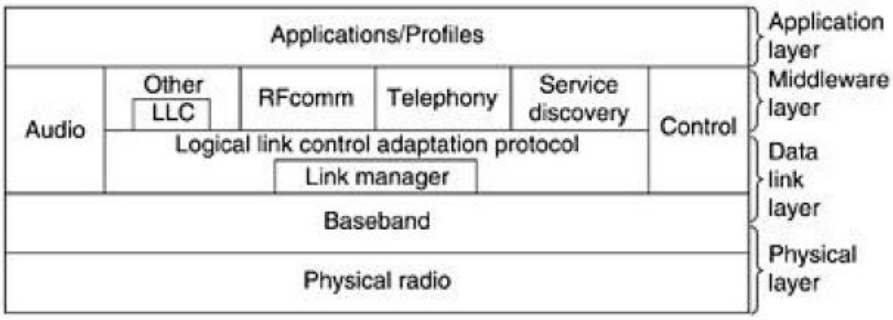

Figure 2-7: The 802.15 version of the Bluetooth protocol architecture.

- Class 1 has a range around 100 meters and uses 100 mW power. It can be used for Bluetooth Access Points.
- Class 2 has a range around 10 meters and uses 2.5 mW power. It can be used for PCs and portable devices.
- Class 3 has a range around 1 meter and uses 1 mW power. It can be used for low power devices.

Bluetooth Baseband Layer

This layer is the most similar layer to the MAC layer in Bluetooth.

Physical Characteristics

The aim of this layer is to map logical channels to physical ones. As we stated earlier, the logical channel is selected among 79 available RF frequencies in the 2.4 GHz band and pseudo-random hopping sequence is used to select the channels. The nodes that use the same sequence form a piconet. Since 1600 hops/sec is used, time is divided into slots of length 625 μsec each by using the clock of the master. Master and slave transmit alternatively such that master uses even numbered slots and slaves use odd numbered slots. Data is transmitted using packets and a packet has a size of 1, 3 or 5 time slots. Frequency hopping timing gives 250-260 slots for settling of radio circuits and 366 bits are used for the packet. If 5 slot packets are used, after settling 2781 of 3125 bits are available for the packet which is much more efficient than single slot packet.

Logical Links

The most used two types of logical links(channels) can be listed as follows:

- **Asynchronous connection-less(ACL):** This type of link is for data communication. Any slave has ACL link by default and a slave can have only one ACL link. This link can be used for symmetrical and asymmetrical asynchronous services. For early Bluetooth technology throughput can be 723.2 Kbps in one direction and 108-433 Kbps for symmetrical flows. Frames can be lost and have to be retransmitted. A slave can only transmit after having received a packet from the master.
- **Synchronous connection oriented (SCO):** This type of link is used for voice communications such as telephone connections. This channel is allocated a fixed slot in each direction. This link is symmetric and can send up to 64 Kbps in each direction. Since time is critical, retransmission is not possible but forward error correction(FEC) can be used for high reliability. There can be at most 3 SCO links in a piconet.

Packets

There are three types of packets.

- **Control Packets:** These are the packets used to carry information about clock, frequency hopping, link establishment or connection requests. There are 5 of them ID, POLL, NULL, FHS and DM1.
- **Synchronous Packets:** These are the packets used in SCO links and they carry the voice. There are various types such as DV, HV1, HV2, HV3.
- **Asynchronous Packets:** These are the packets used in ACL links and they carry data. There are various types such as DM1, DM3, DM5, DH1, DH3, DH5 for different packet sizes.

Error Control

For error detection 2 fields can be used. Error control field(CRC) is 16 bits long and it is at the end of the data body. Error checking is done by calculating CRC pattern of the payload. Header Error Check(HEC) is 8-bit CRC and used to check header in the same manner. After detecting errors, there are 3 types of Forward Error Correction(FEC).

- In 1/3 FEC Coding each bit is repeated 3 times and decoding is done by calculating average of three bits.
- 2/3 FEC Code is a Hamming Code with(15,10) with hamming distance 4.
- ARQ(Automatic Repeat Request) In some packets, data field is retransmitted until a positive acknowledgment or timeout has reached.

Link Manager Protocol

Link Manager sets up and configures the link. Link Manager Protocol is the set of rules to discover and communicate with other devices. There are several states of a bluetooth device such as standby, page, inquiry, connection and park. Except inquiry, these states are managed by LMP. Here, we will explain some of the LMP procedures in detail.

LMP_Sniff_Req, LMP_Unsniff_Req

In the sniff mode, a node gets into a slower cycle and saves energy. A master or slave can request for a sniff mode. If a link is in the sniff mode, only the master can begin a transmission.

LMP_Host_Connection_Req, LMP_Setup_Complete

A connection can be established by sending a paging message on the page scan channel. If the master knows the address of the slave, connection is established. If not, the following procedure is followed.

- If the master does not know the address of the slave to communicate, it is in inquiry state and sends a signal if the node needs a connection. Then the node is in inquiry scan.

- The slave goes into inquiry response substate. The response consists of MAC address of the slave. After the response, slave goes into page scan state and then standby state to listen periodically for the messages. After master get the response it goes to page state and stores the address of the slave.
- If the master wants to establish a link with this slave, it returns a response message to the slave. If the slave sees a message targeting to itself, it goes into Slave response substate and sends its access code to the master.
- When master gets slave's access code it goes into Master response state and returns a packet to synchronize the slave with itself. Then the master goes to connected state.
- Once the slave receives the message, it goes to connection mode, too. Then the master sends a poll message to check the connection and awaits a reply packet.

LMP_AU_RAND, LMP_SRES

With these messages, authentication is done. The master node sends LMP_AU_RAND message that contains a random number(the challenge) to the recipient. After a specific algorithm recipient sends LMP_SRES message and successful pairing of two devices is achieved. Once the node is connected to the master, it can be in the following states:

- Active mode is the mode that a slave is connected and listening the traffic.
- Hold mode is the mode where only synchronous links are active.
- Park mode is the mode where a slave cannot communicate with the master. It is only active at certain slots to synchronize with the master and at the other slots it is sleeping.
- Sniff mode is for energy saving and it can affect ACL links since they are asynchronous.

Logical Link Control and Adaptation Protocol(L2CAP)

This layer has three major functions:

- First, it accepts 64 KB packets from the upper layers, segment them for transmission and reassemble them for upper layers.
- It is responsible for multiplexing and demultiplexing of multiple packets, i.e. determining which upper layer protocol to hand the packets after reassembly.
- It is responsible for quality of service requirements.

Middleware Layer

RFCOMM protocol in this layer emulates serial and USB ports over the L2CAP protocol. It supports up to 60 open ports and provides reliable data transfer, simultaneous connection and flow control.

Service Discovery Protocol is another protocol in this layer. This is the layer on top of the LMP and service discovery is done after authentication. Seven messages are defined to discover and restore the information from client to server.

Bluetooth Profiles

A profile defines a set of protocol components (SDP, RFCOMM) necessary for the set up of correct and communicating applications [14]. Since there were different groups working on specific applications and generated their own profiles, there are tens of profiles. Now, we will give the most prominent profiles:

- Generic Access Profile is not a real application but it provides reliable and secure links.
- Service Discovery Profile is used to discover the services other devices have to offer. With Generic Access Profile, all Bluetooth devices are expected to implement this profile.
- Serial Port Profile emulates a serial line.
- Generic Object Exchange Profile defines the server-client relationships for the data to be transferred.
- LAN Access Profile allows the device to connect a fixed network.
- Dial-up Networking Profile allows a notebook to connect to a phone without wires.
- Fax Profile allows wireless fax machines to send and receive data from phones.
- Cordless Telephony Profile connects the handset of a cordless telephone to the base station.
- Intercom Profile allows two telephones to be used as walky-talkies.
- Headset Profile allows hands-free communication with a headset.
- Object Push Profile allows to exchange simple objects between two bluetooth devices.
- File Transfer Profile is a more general file transfer facility.
- Synchronization Profile provides synchronization of a computer and a PDA.

2-3-2 ZigBee

One can categorize WPANs as low data rate, medium data rate and high data rate. Bluetooth was an example of medium data rate WPANs. To supply low data rate specifications, IEEE 802.15.4 is designed and it specifies physical and MAC layer protocols. ZigBee is the architecture developed on top of the IEEE 802.15.4 reference stack model and takes full advantage of its powerful physical radio layer [14]. Thus, Zigbee provides security, discovery, profiling kind of services on top of IEEE 802.15.4 standard. ZigBee is specified to address low-cost, low-power, lightweight wireless network solutions. By providing long battery life for devices in the networks, it is especially used in wireless sensor networks.

The network layer of ZigBee supports three different topologies: star, tree and generic mesh networks. The commonly used one is mesh topology which enables reliability and long range. In all of the topologies, there should be one coordinator device. There are two kind of capacities of a device: a full-function device (FFD) and a reduced-function device (RFD). FFD can be a coordinator while RFD can only be a slave or normal node.

Physical Layer

There are 27 channels defined in total. 16 of them are on the 2450 MHz ISM band, 10 on 915 MHz and 1 on 868 MHz. 2450 MHz physical layer generates 250 Kbps while 915 MHz and 868 MHz generates 40 Kbps and 20 Kbps, respectively. Unlike Bluetooth, IEEE 802.15.4 uses DSSS(Direct-sequence spread spectrum). On the 2450 MHz band offset quadrature phase-shift keying(O-QPSK) modulation technique is used while on the other bands binary phase-shift keying(BPSK) is used. These phase-shifting techniques minimize power consumption and reduce complexity. Another property of physical layer is energy detection(ED). It can sense the energy level of each channel and it is used for measurement of link quality.

MAC Layer

Channel Access

In IEEE 802.15.4, there are two different channel access methods depending on the system if it supports beaconing or not. The systems that supports beaconing use superframe structure. A superframe starts with a beacon at the first time slot and followed by contention access period (CAP) and contention free period (CFP) in which guaranteed time slots (GTS) are available. And then, there is a inactive period for the device until the next beacon. The systems using superframe structure employs slotted CSMA/CA for multiple access and they can sleep during the inactive period while the other systems employ unslotted CSMA/CA and they should always be active. These are used in CAP.

For low-latency required applications, beacon-oriented systems can use GTSs for transmission. In these time slots CSMA/CA is not used and channel access is guaranteed.

Transfer Models

There are three different data transfer model: coordinator to device, device to coordinator, device to device. In beacon enabled mode, data transfer from coordinator to device starts with a beacon indicates the pending data. Then device sends data request, and coordinator sends acknowledgment message followed by data. The device can send optional ack message. From device to coordinator, device listens for the beacon. When it finds the beacon, it sends the data. The coordinator can send optional ack message to the device. From device to device, the transmission is done by simply sending data to the targeted peer and getting optional ack message back. In all those data transfers, channel is accessed by slotted CSMA/CA. In non-beacon mode, all transfer models are the same without the beaconing. In this case, channel is accessed by unslotted CSMA/CA.

Association-Deassociation with PANs

Association starts with an active or passive scan of a device. A device can either send an active scan message to see the devices in its neighborhood or it can listen beacons from other devices. After scanning period, a device can send an association request to the PAN it wants to join and it gets an address from the corresponding coordinator and joins the network. Deassociation can be done either by decision of a coordinator or device itself. In either case, device and coordinator removes all references of each other. When a device associates with a PAN, it has to synchronize with the coordinator. In beacon-enabled systems,

synchronization is performed by receiving and decoding beacon frames. In the other systems, synchronization is performed by polling the coordinator for data. Synchronization is updated in every transmission. When timeout occurs, device is considered as an orphan and orphan notification commands are sent for re-synchronization.

Frame Structures

There are four types of frames: beacon, data, acknowledgment, control. The general MAC frame structure is shown in Figure 2-8. The descriptions of all types of frames and all the fields are beyond the scope.

<table>
<tr><td>Size in bytes 2</td><td>1</td><td>0/2</td><td>0/2 or 8</td><td>0/2</td><td>0/2 or 8</td><td>Variable size</td><td>2</td></tr>
<tr><td rowspan="2">Frame control</td><td rowspan="2">Sequence number</td><td>Destination PAN ID</td><td>Destination address</td><td>Source PAN ID</td><td>Source address</td><td rowspan="2">Frame Payload</td><td rowspan="2">FCS</td></tr>
<tr><td colspan="4">Address fields</td></tr>
<tr><td colspan="6">MHR</td><td>MAC payload</td><td>MFR</td></tr>
</table>

Figure 2-8: General Frame Structure in ZigBee.

As we stated, these were the layers standard by IEEE 802.15.4. The key definition in ZigBee is called the ZigBee Device Object (ZDO), and addresses three main items: service discovery, security and binding [14].

ZigBee Device Object

In ZigBee networks, data is abstracted as much as possible into Key-Value Pairs (KVPs). The coordinator has a master look-up table known as a binding table that lists which nodes are interested in a particular KVP [15]. The process of securely connecting endpoints of the KVPs is known as binding. It is simply the ability to match two or more devices together based on their services.

In a ZigBee application, KVP clusters are grouped to achieve a particular task. These groups are called profiles. These are almost the same as the Bluetooth profiles. If you cannot find an appropriate profile for your application, you can write your own profile. Service discovery is facilitating the procedure for locating some services via their profile identifiers.

Although, some security issues are mentioned in IEEE 802.15.4 ZDO complements it for security. At each layer security is applied by encrypting the frame and authenticating when it is received. The authentication is done by security services at ZDO. To authenticate and derive a shared secret, link establishment (link key) is used. After security handshaking of two devices, a secure transmission can be achieved.

Another component of ZDO is network manager which is implemented in the coordinator and it decides which PAN to connect. In the creation of a new PANs, it decides the coordinator and slaves and topology of the network. It also connects different PANs by applying routing algorithms.

2-3-3 Wi-Fi

Like ZigBee being developed on top of IEEE 802.15.4 specification, Wi-Fi is a trademark of the Wi-Fi Alliance developed on top of IEEE 802.11 WLAN standards.

Wi-Fi supports three different operational modes. In the infrastructure mode, the wireless network consists of at least one access point(AP) connected to a fixed network and client stations connected to the access point. This mode is based on cellular architecture, each cell is called BSS (Basic Service Set) consisting of one AP. Each BSS is executing same MAC protocol for multiple access within the range. Second mode is ad-hoc mode often referred as peer-to-peer mode. In this mode there is no need to an AP and each station can establish a communication with any other station in the cell. These cells are called IBBS(Independent Basic Service Set). This mode allows to create quickly and simply a wireless network where there is not fixed infrastructure or where such an infrastructure is not necessary for the required services (hotel room, conference centers or airport) [14]. The third mode is Mesh Configuration which is the hybrid configuration of both infrastructure mode and ad-hoc mode.

Physical Layer

We have seen two different physical layers before. Frequency Hopping Spread Spectrum (FHSS) based physical layer is used by Bluetooth. Direct Sequence Spread Spectrum (DSSS) is used by ZigBee. DSSS-based physical layer supplies high throughput and broad coverage while FHSS-based physical layer is suitable for highly multipath environments. In Wi-Fi physical, layer DSSS is used to achieve high throughput. In this physical layer ISM band 2.4-2.4835 GHz is used and 14 channels of 22 MHz separated by 5 MHz are defined. In early versions, OFDM(Orthogonal Frequency Division Multiplexing) technique is used for modulation and 6-54 Mbps throughput could be achieved. Typical radio range of Wi-Fi is 100 meters and with 6-54 Mbps data rate, Wi-Fi is more suitable for indoor high rate applications.

For the LLC layer, Wi-Fi uses same properties that an LLC 802.2 layer and allows to connect a WLAN to any other LAN of the IEEE family.

MAC Layer

Channel Access

In WI-FI, superframe structure is similar to the one in ZigBee. The superframe starts with a beacon, then contention free period (CFP) and contention period(CP) follows. In CFP, point coordination function(PCF) is selected for transmission which is similar to GTSs in ZigBee. This function is conceived for the data transmission with time constraints such as the real-time traffic (voice, video). Basic access method in WI-FI is applied during the CP and is called distributed coordination function (DCF). CSMA/CA protocol is used as an access method for DCF. To solve hidden and exposed node problems, it is supplemented by two signals Request to Send(RTS) and Clear to Send (CTS) as we explained before.

Association, Reassociation and Disassociation

The active and passive scanning process explained for ZigBee is the same for WI-FI. Either

a station searches for a access point(AP) or an AP periodically sends a beacon. When a station wants to join a BSS, it chooses it according to signal power, signal quality and network load. After data is exchanged for association, authentication is preceded for reliable data transmission. The synchronization is again sustained by periodically received beacons from the AP. Disassociation can be carried out both up on request of AP or the station itself. Reassociation happens in several cases which are roaming, high traffic, change of the radio environment or high error rates. In high traffic, reassociation brings load balancing. The principle of roaming is similar to handover in GSM. In mobile wireless networks, a client can move out of the range and should reassociate to another AP by concerning the same criteria as association. The process of dynamic association and reassociation gives the network ability to cover a wide range area.

Security

The basic security issues of Wi-Fi follows WEP (Wired Equivalent Privacy) standards. WEP applies following security services

- The shared key authentication is used. In the association process of a station, AP sends a random number, station encrypts random number and sends back to AP. After AP decrypts the number and matches the key, association is completed.
- WEP Integrity: When a frame is sent to a station, station appends 32 bit Integrity Check Vector (ICV) to the frame.
- WEP Confidentiality: Since the same shared key is used in every transmission, an additional 24 bit Initialization Vector(IV) is appended to each frame. Since IV is plaintext, anyone sniffing the WLAN can see it. Thus it should not be used twice. However, with 24 bits, 16,777,216 possible numbers can be generated and in 5000 transmissions, possibility of generating the same number is 0.50. So, anyone can listen and capture the data if the same key is generated.

Since WEP supplies very unsecured security services, latter versions of Wi-Fi uses WPA(Wi-Fi Protected Access) or WPA2. In these standards, 256 bit key is used and since these keys are vulnerable to cracking, further protection is applied.

Up to now, we have seen three different wireless technologies: Wi-Fi, Bluetooth and ZigBee. In Figure 2-9 we can see the comparison of these wireless technologies.

2-3-4 Zero Configuration Networking

Communication across the planet is supplied by TCP/IP without worrying about the type of physical connection such as Ethernet, wireless, etc. This is the inspiration of solving the local communication problem. For local purposes, TCP/IP is the way too powerful and complicated technique. For instance, assume there is a user wants to set up a local network with his camera, printer and computer. To assign an individual IP he has to set up Dynamic Host Configuration Protocol (DHCP) or do it manually. To assign local names, he has to

	Wi-Fi	Bluetooth	ZigBee
Frequency bands	2.4GHz	2.4GHz	2.4GHz, 868 / 915 MHz
Stack size	~1Mb	~1Mb	~20kb
Raw data rate	11Mbps	1Mbps	250kbps (2.4GHz) 40kbps (915MHz) 20kbps (868MHz)
Number of channels	11 – 14	79	16 (2.4GHz) 10 (915MHz) 1 (868MHz)
Data types	Digital	Digital, Audio	Digital, Key-Value Pairs
Inter-node range	100m	10m – 100m	10m – 100m
# of devices	32	8	255 / 65535
Power requirements	Medium – hours on one battery	Medium - days on one battery	Very low – years on one battery
Current market penetration	High	Medium	None
Architectures	Star	Star	Star, Tree, Cluster
Best applications	Internet inside buildings	Computer & phone peripherals	Low-cost control and monitoring

Figure 2-9: Comparison of Different Wireless Technologies.

set up Domain Name System services (DNS) or do it manually. Thus, he is expected to be a network expert to set up a simple local network. It is analogous to being expected to be a mechanic to drive a car. To solve this problem a new software was implemented. Zero Configuration Networking - Bonjour, as Apple calls it - provides a three-layer foundation to enable hardware and software makers to produce great products compatible to set up easy networking [16]. The first layer is getting an IP address when there is no DHCP server. Second layer is automatic resolution and distribution of computer Host names when there is no DNS server. Third layer is automatic discovery of network services.

IP Addresses without DHCP

Each device on a network needs at least one unique IP address. There are three ways of obtaining an IP address. First one is manual addressing by a network administrator. However, with this technique one should be expert on networks for proper configuration. Second method is having a DHCP server assigning IP addresses to the nodes. When there is no DHCP server, IP addresses can be self-assigned, in other words each device is responsible for choosing its own IP address which is called link-local address configuration and zeroconf protocol is using this technique.

In link-local addressing there are 65024 addresses available. IP obtaining procedure is as follows:

- New host randomly selects IP-address U out of 65024 available addresses
- Broadcasts a probe message: Who is using address U? For probing, Address Resolution

Protocol(ARP) is used. In this protocol you broadcast an ARP message to reach desired IP.

- If you get an ARP reply it means another host is claiming the same address. In this case, the host chooses another address.
- If there is no ARP replies, it means that address U is available.
- However, there can be message loss and delay of messages. Depending on the protocol that is used there can be more probing. For instance, IPv4 is using 4 probes.
- If none of the probes you get an ARP reply, then use U as an IP address.
- Then, you announce your address to the network using ARP.

However, there is always a small probability of choosing an existing IP address. In that case, late conflicts happen. The reason for this may be network topology changes, misbehaving hosts, etc. Whatever the reason is, the conflict has to be handled. Dynamic Configuration of IPv4 Link-Local Addresses (RFC 3927) allows the host to handle the conflict in two ways. The first way is backing down and choosing another IP. If it has open TCP connections which it does not want to lose, second way is sending ARP announcement asserting its own ownership of the address. In second way, when both of the hosts does not back down, finally one of them lose open connections and back down.

Names without DNS

IP addresses may change over time, and network location and IP addresses are not human-friendly form to remember or recognize devices. So, it is easier to use text-based names instead of IP addresses. In global networking domain name system(DNS) is used. However, to use DNS, one should set up DNS servers which is very costly. When you need names that are only valid for local links, link-local Multicast DNS provides a simple solution.

In normal DNS, namespace is URLs such as *http://tas.tudelft.nl* in which *http* identifies the protocol, *tas* is a node in *tudelft.nl* domain and *tudelft* is node in the *nl* domain. When someone enters this URL, DNS server is connected, the name is resolved and searched hierarchically and the node is found. In Multicast DNS, local is used in the namespace - *ali.local* - to distinguish it from the global addresses. In this case, the name *ali* is sent to all nodes in the network and when a node sees a query for its own name, it answers the query. Instead of a DNS server, additional software can handle this property.

Unlike the IP assignment, the host name is set manually. Once the local name is created, a local Multicast DNS address record maps the names to the IP addresses. To check the uniqueness, again probing technique is used. This time three queries are sent and if there is no conflict, uniqueness is verified and the host announces its name to the network. When there is a conflict, a prompt message is sent and user is asked to change the host name.

Service Discovery

The last technology that Zeroconf uses is DNS Service Discovery(DNS-SD) which lets you know the available services on the network without knowing the names or IP addresses of devices. To use this service there should be link-local Multicast DNS or global Unicast DNS should be available. The service discovery software has two main responsibilities: enumerating the list of services, translating from service name to the IP address of the node. In order to be more useful, Zeroconf service type gives both information about the service and which protocol has to be used. For instance, *_ipp* encodes both printing and via Internet Printing Protocol.

DNS protocol family already defines a record type of SRV for service discovery. In Zeroconf, we have DNS pointer(PTR) each pointing to different SRVs. By performing a PTR lookup for a name of a service, you can get all instances and choose one of the services. In order to retrieve the information for connecting DNS-SD TXT record can be read and all information such as host name, IP address or the port number that corresponding service resides, etc can be reached. In the existing case of a Unicast DNS server, same concepts can be used hierarchically to reach entire world.

Chapter 3

Implementation of the Distributed Robotics Library

In this chapter, the design and implementation of the Distributed Robotics Library (DRL) will be introduced. In section 3-1 the design of the DRL will be explained. Section 3-2 explains the main functions in the library. In section 3-3 connection establishment and message transfer between two agents are explained by using a UML sequence diagram.

3-1 Design of the DRL

As explained earlier, the DRL is implemented to set up a communication network in any distributed robotics system to allow decentralized control. The design of the DRL follows the 7 layer Open Systems Interconnection (OSI) model with some slight differences. Before explaining the design of the DRL layer by layer, we will give some basic information about the design.

The DRL is implemented in C++ programming language and consists of two main classes:

- *DistributedRoboticsLib* class takes care of the identification of agents, establishment and maintenance of the connections in the network.
- *DistributedRobotCommLib* class takes care of the private communication between two specific agents.

In Figure 3-1, we give a diagram that shows the ownership of the objects that are used in Figure 3-2. As we see, *Robot1*, *Robot2* and *Robot3* are objects of *DistributedRoboticsLib* and *RobotComm*'s are objects of *DistributedRobotCommLib*. From here on, we will denote the objects of *DistributedRoboticsLib* as *Robot* and the objects of *DistributedRobotCommLib* as *RobotComm*.

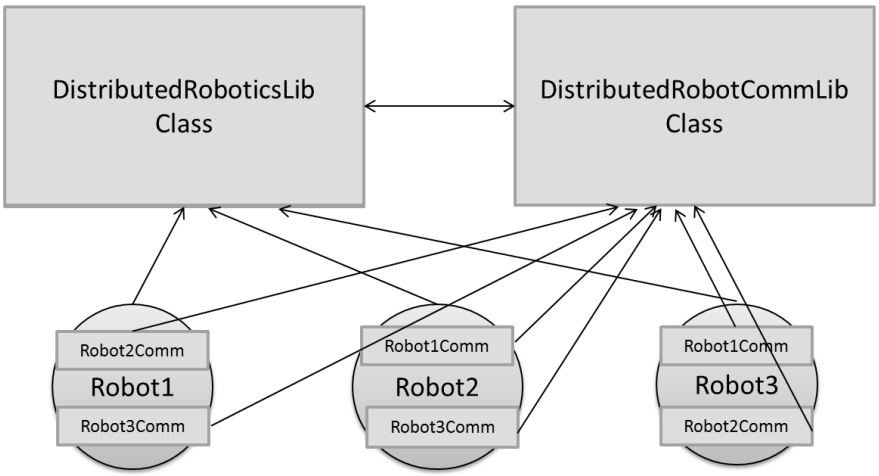

Figure 3-1: Relationships of the objects with the DRL Classes.

In Figure 3-2, communication relationships of objects in a network consisting of 3 agents are shown. If there are n agents in the network, then, there will be n *Robots*, 1 for each agent. However, for each Robot, $n-1$ *RobotComms* will be introduced to take care of communication with the other $n-1$ *Robots*. So there will be $n \times (n-1)$ *RobotComms* in the total network.

3-1-1 Layers of Distributed Robotics Library

The DRL is designed to be used in networks with mobile agents. Thus, the physical layer transmission should be wireless. For the physical layer(layer 1) and the data link layer(layer 2) Wi-Fi standards are used. Theoretically, ethernet protocol with an ethernet cable can also be used, but since the robots are mobile, practically it is not applicable. Therefore, the agents which want to use the DRL should support Wi-Fi technology. Wi-Fi provides standard for the lowest two layers of the DRL. It uses WLAN(IEEE 802.11) standard for the physical layer and Carrier Sense Multiple Access With Collision Avoidance (CSMA/CA) technique for the data link layer. These layers and protocols that we used are explained in Chapter 2 in detail. However, detailed information is not needed to understand the DRL.

For this version of the DRL, we use IPv4 protocol for the network layer(layer 3) since it is the most prominent network layer protocol and supported by the PCs that will be used by the robots. However, IPv4 provides more services than we need for a local network and this causes transmission of redundant data along with the core data. In future, more suited network layer protocol can be designed for the DRL. In Figure 3-3, we see first 3 layers of the DRL with the protocols used.

In the DRL, there are three different communication processes. For the identification of the agents within the network, an automatic advertisement process is developed. To establish connection between agents, a connection request/accept process is implemented and for the actual data transfer between agents, a server/client communication process is used. The first 3 layers explained in the previous paragraphs are common in all these three processes.

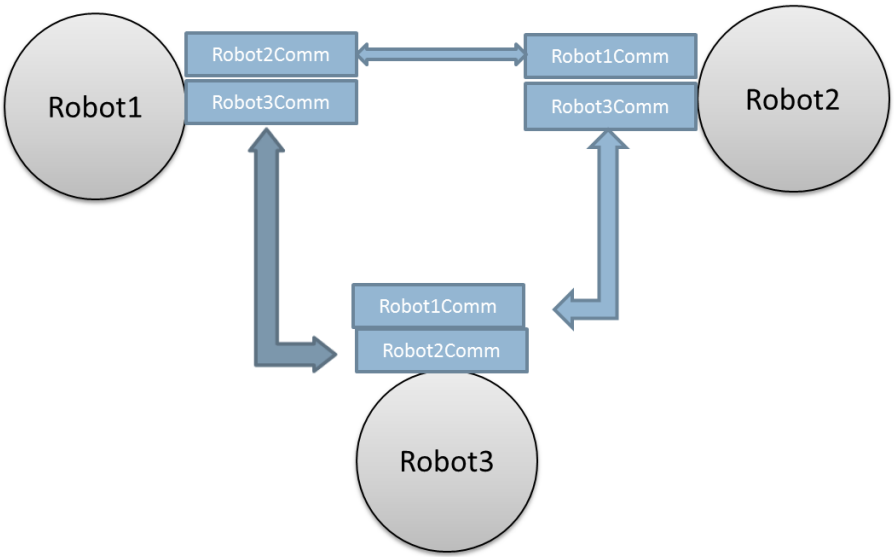

Figure 3-2: DRL Objects of a network with three robots.

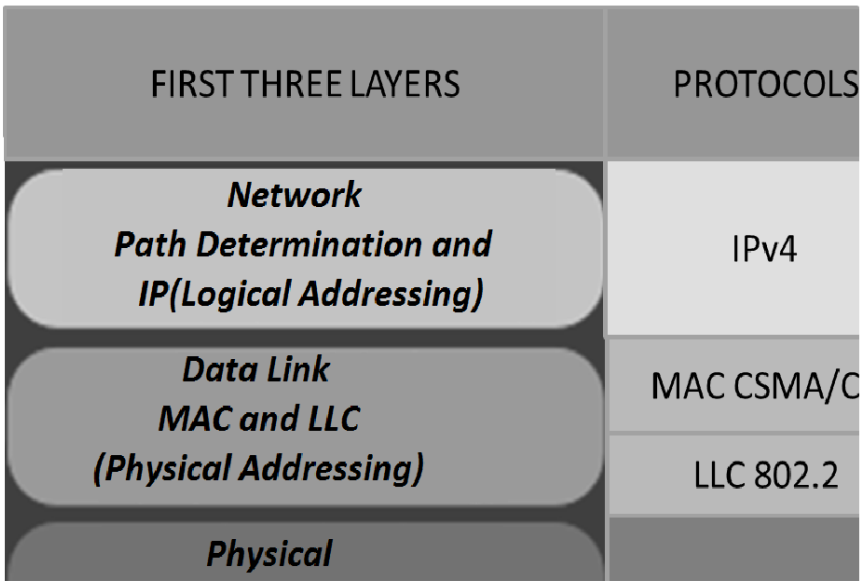

Figure 3-3: Protocols used in the first three layers of the DRL.

However, the protocols used for the upper layers are slightly different. Now, we will explain these processes in detail:

Advertisement Process

Advertisement process is used by a *Robot* to introduce itself to the network and identify the other *Robots* in the network. Every *Robot* in the network has a unique HOSTNAME and (not necessarily unique) PORT number. While HOSTNAME is used to identify the *Robot*, PORT number is used to identify the communication channel that *Robot* is using. Advertisement information consists of these two variables of the *Robot*. In order to notify other *Robots* about its existence in the network, the *Robot* sends the advertisement information each second to the whole network.

Transport Layer of Advertisement Process: For the Transport Layer of advertisement process, User Datagram Protocol (UDP) is used since it uses a simple transmission model without implicit hand-shaking dialogues. Without the need of the address of the receiver, UDP allows us to send the advertisement information to the whole network. Since we are sending the advertisement each second, we need fast transmission of the data which is supplied by UDP. Although UDP does not provide a reliable service, loss of an advertisement is not critical in a network since it is sent each second.

As we explained in Chapter 2 the agents can use software interfaces which is not specified in OSI model. In advertisement process and the other two processes, we use a very powerful interface between Layer 4 and 5. *Practical C++ Sockets*, which provides wrapper classes for a subset of the the Berkeley sockets application programming interface (API) TCP and UDP sockets and works on both the Unix (tested under Linux, RedHat 7.3 with gcc) and Windows (tested under Win2K with Visual C++ 6.0) platforms, are used in our library. Practical C++ Sockets provides a very simple C++ interface to sockets[17] and the whole software is freeware.

In high level protocols such as TCP/IP it is difficult to separate the upper 3 layers of the OSI model. For this reason, we will call the rest of the protocol as "Application Layer" which includes the OSI Application Layer, Presentation Layer and the Session Layer.

Application Layer of Advertisement Process: Each *Robot* using the DRL is able to send and receive advertisements by using multi threading. Sending unit and receiving unit are completely independent from each other. In sending unit, first data to transmit is determined which is comprised from unique HOSTNAME and PORT number of the *Robot*. As stated earlier, unique HOSTNAME is used to identify the *Robot* and PORT number for connection establishment which will be explained later . In general, unique IP address is used to identify the agents. However, mobile agents generally do not use static IP addresses which means IP addresses can change in every reconnection to the network. Hence, using IP addresses would require some adaptive techniques to handle the changing IP's. Instead, we decided to use HOSTNAME, which is a label assigned to a device connected to a computer network, for identification.

As a requirement of the Presentation Layer we have to provide common representation of data to be transferred between application entities. In this version of the library, "Practical C++ Sockets" interface allows us to send characters directly without any encoding. In the future, if lower levels of the DRL are implemented, it might not be possible to use characters that easily. In that case, best standard would be American Standard Code for Information Interchange (ASCII) to represent data because most of the modern character encoding schemes are based on ASCII. After representation of characters, a representation structure of data should also be defined in this layer. We used the most prominent data encoding standard, Extensible Markup Language (XML) standard for its simplicity and generality. XML is a flexible way to create self-describing data and to share both the format and the data on the World Wide Web, intranets, and elsewhere[18]. Since XML is simple and easy to use, XML-based formats have become the default for most office-productivity tools. Note that, presentation layer is common in all three of the processes. After encoding the advertisement message with XML, it looks like this:

```
<HOSTNAME>ALPHA1</HOSTNAME>
<PORT> 51000 </PORT>
```

In order to create XML messages and retrieve data from XML messages easily, we need a XML processor which is called XML Parser. Among many freeware C++ XML parsers, we chose CMarkup because of its simplicity, ease of use and speed [19]. This library provides all the functions we need, to create and parse XML messages.

After creating the advertisement message in sending unit, the robot creates a UDP socket at a preshared port number ADV_PORT which is known by all the *Robots*. Then *Robot* sends the advertisement message to the IP address "255.255.255.255" through the UDP socket. This specific IP address enables us to send the message to the whole subnet that the agent is connected. Although there is a unique broadcasting IP for every network, this IP works for every network. The *Robot* sends the same message in each second to the whole network. While maintaining real-time existence information of the *Robot* in the network, unreliability of UDP protocol is also solved by sending out the advertisement every second.

In receiving side of the application layer, there is a UDP socket that is listening to ADV_PORT. When a message is detected, the IP address and PORT number are parsed and the list of *RobotComms* is checked whether the advertising *Robot* already exists or not. If not, it means that the *Robot* is new in the network and a *RobotComm* is created for this *Robot* and added to the list. If it already exists, then corresponding *RobotComm* is informed about its existence not to remove itself. We will explain removing conditions later in detail. After the advertisement is received and processed, the Robot starts to listen advertisements again. We can see the advertisement process in a network with two agents in Figure 3-4 and flow chart of the process on *Robot1* in Figure 3-5. When *Robot2* joins the network at $t = 0$, it sends the advertisement to the whole network. *Robot1* receives the advertisement and adds *Robot2Comm* to the list. Before $t = 1$, which is 1 second after *Robot1*'s last advertisement, *Robot1* sends advertisement and *Robot2* adds *Robot1Comm* to its own list. At $t = 1$, *Robot2* sends the advertisement. Since *Robot2Comm* is already in the list it is informed to stay alive.

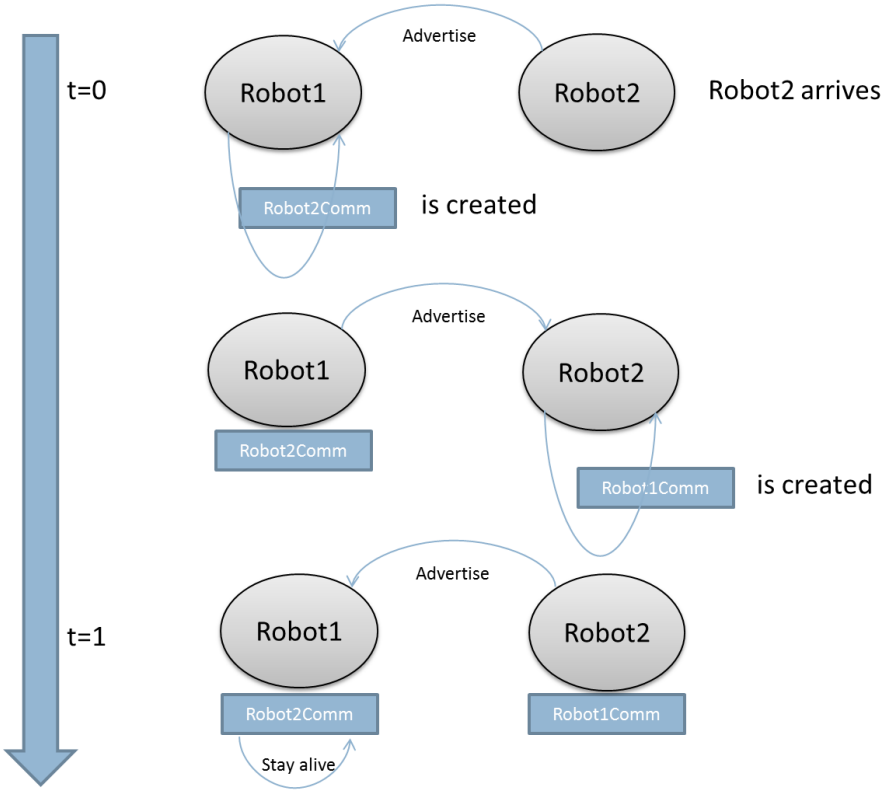

Figure 3-4: Advertisement process between two agents.

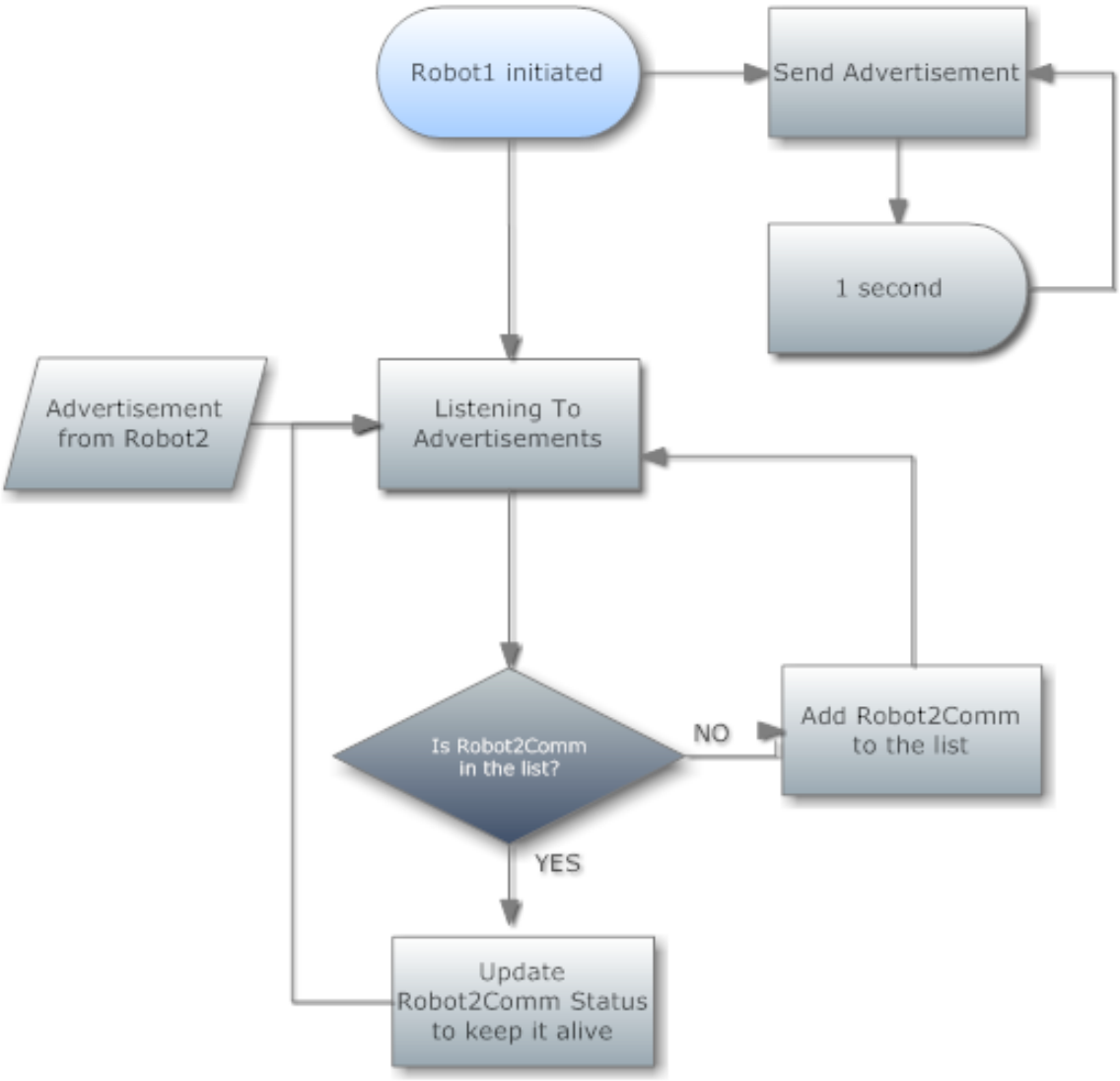

Figure 3-5: Flow Chart of Advertisement Process on Robot1.

Connection Request/Accept Process

Connection request/accept process is used to establish communication between two *Robot* entities. A *Robot* in the network can send a connection request to a specific IP address. The connection request process starts with finding the *RobotComm* with the IP address. If there is no *RobotComm* with that IP address or there is already a connection with that *RobotComm*, then the connection request is ignored. Otherwise, the *Robot* creates a TCP socket to send a connection request to the target *Robot*. If the TCP socket cannot be created for any reason, the process is repeated until a the connection request is sent. If the socket is successfully created, then the advertisement information is sent to the target *Robot*, to indicate the source of connection request. Then, the *Robot* waits for data until it comes.

Right after its creation, each *Robot* starts to listen connection requests by creating a TCP server socket at its specific port. Whenever, another *Robot* tries to send a connection request by creating a TCP socket with the IP and port of the listening *Robot*, it accepts the socket and waits for the connection data. In this version of the DRL, there is no condition for accepting or refusing a connection request. Hence, every connection request is by default accepted. After the client *Robot* sends its data(as explained in the previous paragraph) the data is parsed and IP address and port number of client is determined. Then, a randomly created port number is sent to the client *Robot*. If there is no *RobotComm* with this IP address, which is the situation that the server *Robot* did not receive client's advertisement yet, the *RobotComm* with this IP and port is added to the list. Then, this *RobotComm* creates a TCP server socket on the new port to start the communication as a server and waits for the client to connect. After the further communication is handed over the *RobotComm*, the *Robot* starts to listen connection requests again.

After client *Robot* gets the data that it was waiting for(the new port number), the *RobotComm* which is responsible for the communication with the server Robot is found and that *RobotComm* creates a TCP socket with server's IP and new port to start the communication as a client. Then the socket is deleted to end the current session. In Figure 3-6 we see simple connection request/accept process in 7 steps. After *Robot2* sends the connection request, *Robot1* detects the connection request, determines the IP of the requester and finds corresponding *RobotComm* (*Robot2Comm* in this case). Then, *Robot1* creates a new port and sends it to *Robot2*. Then, *Robot2* finds *Robot1Comm* in its list. After *Robot2Comm* of *Robot1* creates the server, *Robot1Comm* of *Robot2* connects to the server and becomes the client. In Figure 3-7, we see the flow chart of the process running on *Robot1* side.

Server/Client Communication Process

Server/Client communication process is at *RobotComm* level and takes care of the actual data transfer between agents. Although we call this process as server/client communication, both server and client are able to send and receive data and there is no difference as communication partners. After creation of sockets at both sides, both sockets starts to listen for incoming data.

As we explained earlier, the messages in this communication are also encoded with XML. When a *Robot* wants to send a message to another *Robot*, first communication is provided between *Robots* if there is not. Then, the corresponding *RobotComm* is found and this *RobotComm* sends the message over its socket. Since the corresponding *RobotComm* of the other

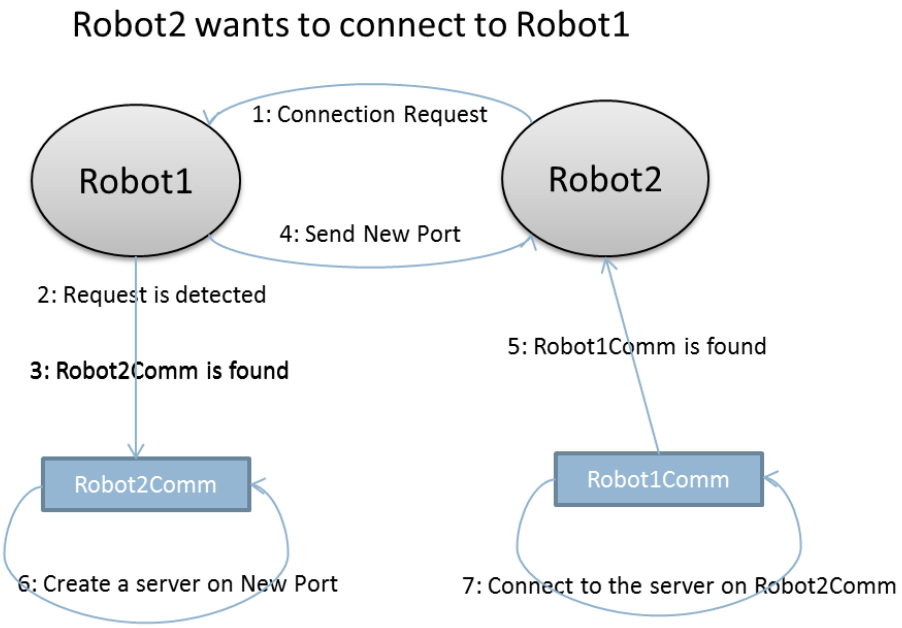

Figure 3-6: Connection request process.

Robot is already listening messages, it receives the message, parses the message and applies the action in the message. Then, *RobotComm* starts waiting for another message. This communication is continuous until one side is not available(outside the range, shut down, sleeping, etc.) or wants to stop communication. Under non-availability conditions, the *RobotComm* deletes the socket and removes itself from the *RobotComm* list of the *Robot*. If one party wants to stop the communication, the socket is deleted but the *RobotComm* is still alive for further communication requests.

In the DRL, the handling of the received message is also included in the application layer of the process. After creation of a *Robot*, the user has to provide the pointers of agents functions to the *Robot*. Although the method will be explained in the next chapter, we give some brief information about the process. In a C++ file associated with the DRL, the agent has list of its functions and their definitions. To provide a common representation of agent's actions, everything the agent is able to do should be defined as a function which has a fixed structure. So, every message that is sent to robot becomes a function call. The Robot keeps pointers to these functions in a list. Whenever a message comes and is parsed by *RobotComm*, the pointer to this function is found in the *Robot* and function is called with the provided parameters in the message.

As a simple example shown in Figure 3-8, assume that *Robot2* wants to change the location of *Robot1* to (10,15) by using the *ChangeLocation* function already exists in the list of *Robot1*'s functions. The XML message that is sent to Robot1 looks like:

```
<ChangeLocation>
```

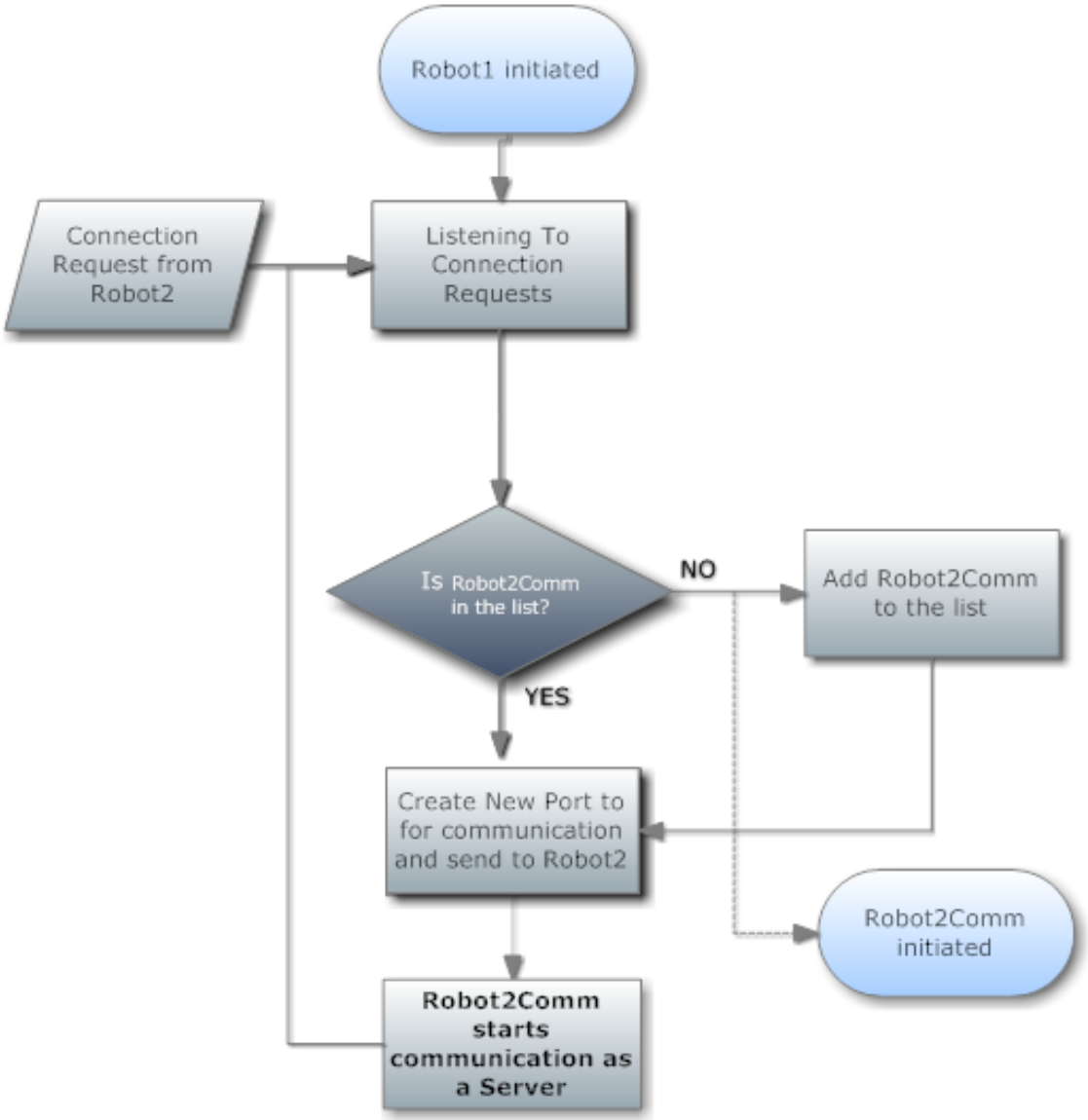

Figure 3-7: Flowchart of Connection Request Process on Robot1.

```
        <X>10</X>
        <Y>15</Y>
    </ChangeLocation>
```

After Robot1Comm of Robot2 sends this message to Robot2Comm of Robot1 which was already listening to coming messages, it receives the message, parses it, finds the Change-Location function of Robot1 and calls the function. ChangeLocation function includes the actions that Agent1 has to do to change the physical location to (10,15). Thus, the task is complete and both RobotComm's keep on listening for further messages. In Figure 3-9, we see flow chart of the process on *Robot2Comm (Robot1)* side.

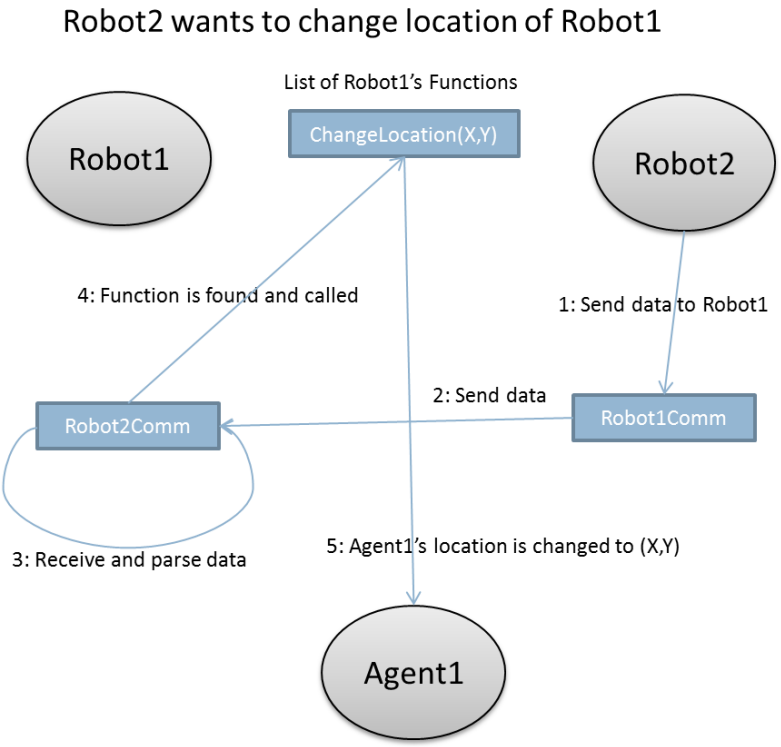

Figure 3-8: Server/Client Communication Process.

3-2 Main Functions of Distributed Robotics Library

Before explaining the functions, some structures, variables and important member variables of both of the classes of the DRL should be presented.

- **message_handler** : This is a structure consists of function name, pointer to that function and a pointer to an object (The significance of third variable will be explained

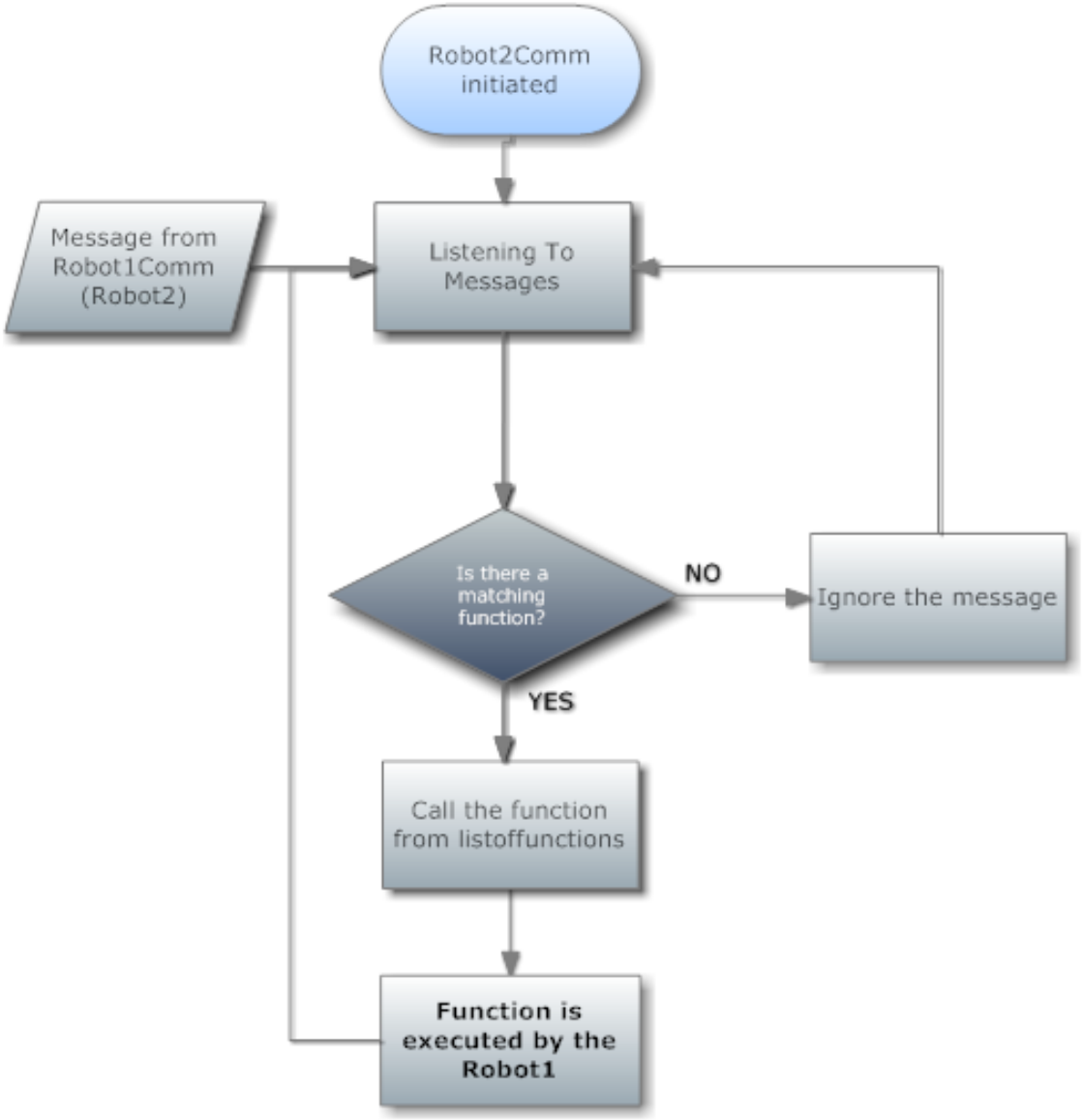

Figure 3-9: Flow Chart of Server/Client Communication on Robot1.

later). A message handler is basically used to keep the pointer of a function of the robot.

- **ADV_PORT** : This is the advertisement port which is used by every robot to send and listen advertisements.

The following variables are member variables of the class *DistributedRoboticsLib*:

- **hostname** : The unique hostname of the *Robot*.
- **port** : The port of the *Robot* which is used to contact it. It does not have to be unique. In our applications
- **info** : The information of the *Robot* which can consist of available functions or member data.
- **function_list** : This is a list of type *message_handler* consisting of all the message handlers of the *Robot*.
- **robot_comm_list** : This is a list of type *DistributedRobotCommLib* consisting of the *RobotComm*'s for the other *Robots* on the network.

The following variables are member variables of the class *DistributedRobotCommLib*:

- **hostname** : The hostname of the *Robot* to be communicated with.
- **ip_address** : The IP address of the *Robot* to be communicated with.
- **port** : The port of the *Robot* to be communicated with.
- **parent_robot** : A pointer to the *Robot* who owns this *RobotComm*.
- **is_connected** : A boolean variable which keeps track of the connection to the corresponding *Robot*.
- **watch_dog** : An integer value decreasing by 1 every second. It is used as a watchdog timer. When it becomes 0, *Robot* removes *RobotComm* from the list and *RobotComm* is deleted.

3-2-1 DistributedRoboticsLib Class Functions

Advertise : Advertise is a thread which starts running with the creation of the *Robot*, it advertises the *Robot* to the network. For the advertisement the hostname and the port of the *Robot* are encoded into XML format which can be seen in the previous chapter. After creation of a UDP socket, the encoded data is sent to the IP address "255.255.255.255" and port number ADV_PORT over this UDP socket. IP address "255.255.255.255" is used to send the data to the all *Robots* on the same subnet. This process is repeated in each second.

ListenToAdvertisements : This is a thread which starts running with the creation of the *DistributedRoboticsLib*, it creates a UDP socket to listen ADV_PORT, and it listens this socket forever. Once it receives a message, it decodes the message and checks if the advertising *Robot* is already in the robot_comm_list. If yes, first it first checks for an IP change. If IP address of the *Robot* has changed, *RobotComm* is removed and added with the new IP. If *RobotComm* already exists with the same IP, it sets the value of watch_dog variable to 3 in order to keep *RobotComm* alive. If not, it creates a new *RobotComm* and adds to the list. Then it goes back to advertisement listening mode.

ListenConnectionRequests : This is a thread which starts running with the creation of the *Robot*, it creates a TCP server socket to listen ADV_PORT, and it waits for the connection to the server forever. Once it detects a connection attempt to the server, it waits for the short information message. In this version of the library, advertisement message is used for the short information message. Once it receives the information message of the robot which is trying to connect, it decodes the message and checks if the corresponding *RobotComm* is already in the robot_comm_list. If not, it creates a new *RobotComm* and adds to the list. Then, it generates a random port number to communicate with the robot and sends this port number to the *Robot* which is trying to connect. Then it calls *StartCommunicationAsServer* method of *RobotComm* with the new port number. Then, it goes back to connection waiting mode.

SendConnectionRequest : This method sends connection request to a particular hostname in the network. First, it tries to find corresponding *RobotComm*. If there is no *RobotComm* with this hostname, connection request is ignored. If *RobotComm* is found, port number of the *Robot* is retrieved and short information is sent to the *Robot*. In case the target robot is not listening the connection requests for the moment, it keeps on sending the connection request until the request is sent. Then, the method starts listening the new port number which will be created by the server *Robot*. Once it receives the new port number, it calls *StartCommunicationAsClient* method of corresponding *RobotComm* with the new port number.

StopCommunication : This robot is used to stop communication with a *Robot*. First it finds the corresponding *RobotComm*. Then, it calls *StopCommunication* method of *RobotComm*. The rest will be handled by *RobotComm* and will be explained later.

SendXMLMessage : This method is used to send an XML message to a particular hostname. First, the corresponding *RobotComm* is found by searching the robot_comm_list. If there is no, then message is ignored. If found, then the message is sent by calling *SendXMLString* method of *RobotComm*. The rest will be handled by *RobotComm* and will be explained later.

CreateMessageHandler : This is one of the fundamental functions of the library. The function simply adds a new message_handler to the function_list. It has three inputs:

- **function_name**: This is the tag name of the callback function. In XML messages this name should be used.
- **function pointer**: This is pointer to the callback function which should be in "ListOfFunctions.h" file.
- **data**: This is a void pointer. In case, the function to be called is a function of another entity inside the agent, a pointer to this entity should be passed to use its functions. In this case, the callback function in "ListOfFunctions.h" is used as an interface. The significance of this input will be explained better in Chapter 4.

3-2-2 DistributedRobotCommLib Class Functions

StartCommunicationAsServer : This function creates a TCP server socket on the new generated port by *ListenConnectionRequests* function of the *Robot*. Then it waits for a connection to the server. Once a client is connected to the server, it creates *HandleTCPClient* thread and sets is_connected to 1.

StartCommunicationAsClient : This function creates a new TCP socket to connect to the TCP server created in the above function. Once it is connected, it creates *HandleTCPClient* thread and sets is_connected to 1.

HandleTCPClient : In this thread, the socket, which was created in the functions above, is waiting for a message. Once it receives a message, it retrieves function name to be called from the message and searches function_list of the *Robot* for a match to the function name. Once it has a match, it calls the function with the function pointer. Then, it goes back to message listening mode. Once it receives an empty message, which means the other party somehow stopped the communication, it removes this *RobotComm* from the robot_comm_list of the *Robot*. Once, an exception is caught from the socket, it means that there is something wrong with the connection. Then is_connected is set to 1 and *HandleTCPClient* thread is killed.

The message format sent and received by this function looks like:

```
<ChangeLocation>
    <X>10</X>
    <Y>15</Y>
</ChangeLocation>
```

Note that *ChangeLocation* is a "setter" function. Hence, there will be no data to be returned back. However, when the function is a "getter" function or any function that returns data, the XML message should include handler information of the sender. Handler information lets receiver know how the sender will handle the data, so that the receiver can send the data back as another XML message. An example of a message with handler information looks like:

```
<GetTime>
    <HANDLER>
```

```
        <FUNCTION>PrintMessage</FUNCTION>
    </HANDLER>
</GetTime>
```

In this message, sender asks for the time of the receiver and tells the receiver that it will handle the returned data by printing it. In this case the reciever's return message would look like this:

```
<PrintMessage>
    <STRING>''Time''</STRING>
</PrintMessage>
```

WatchDog : In this thread, watch_dog variable is decreased by 1 in every second. When it hits to 0, this *RobotComm* will be removed from the robot_comm_list of the *Robot*.

StopCommunication : This function stops the communication by killing *HandleTCPClient* thread and deleting the port. Then, it sets is_connected to 0. Note that, *RobotComm* will still stay in the robot_comm_list of the *Robot* but without connection.

SendXMLString : This method simply sends the XML message provided by *SendXMLMessage* function of the *Robot*. The message sent by this method is handled by *HandleTCPClient* thread of the partner *RobotComm*.

3-2-3 ListOfFunctions Class

Although *ListOfFunctions.h* and *ListOfFunctions.cpp* is not a part of the DRL, it is a file that should be provided by the user who wants to use the DRL. *ListOfFunctions.cpp* is the file that consists of function definitions which act like an interface between the DRL and functions of the agent. The functions should have a fixed signature which is the same as the function pointer in message_handler structure. The signature of the function is as follows:

```
void MoveFirstLeg(CMarkup* xml, void* data, void *S);
```

The first input is the pointer to the XML message that is sent to the *RobotComm*. The second input is the pointer to the entity of the agent from which the real function will be called. It is the third input of *CreateMessageHandler* function that is passed to the function. Third input is a pointer that is passed to the function in *HandleTCPClient* function. It is a pointer to the *RobotComm* which received the message. This pointer is used to send a message back to the sender *RobotComm*.

The functions in the *ListOfFunctions.h* should be structured to do the following tasks:

1. Parse XML string and retrieve the inputs that are needed to call the function.
2. Call the function.
3. If it is a "setter" function, do nothing.
4. If it is a "getter" function, save the output of the function. Retrieve handler information from the XML String and prepare the return message by adding the result of the function.

3-3 Execution of the DRL

In this section, we will explain the execution of the DRL between two agents. UML Sequence diagram is utilized to give a better understanding.

The acronyms used in the Figure 3-10 are as follows:

- H1 : Hostname of Robot1
- P1 : Port number of Robot1
- Robot1 : DistributedRoboticsLib with H1 and P1
- Robot2 : DistributedRoboticsLib with H2 and P2
- R1::SCR : Robot1.SendConnectionRequest(H2)
- R1::SXML : Robot1.SendXMLString(H2)
- R1::A : Robot1.Advertise()
- R1::LCR : Robot1.ListenConnectionRequests()
- R1::LA : Robot1.ListenToAdvertisements()
- Robot2Comm : Robot1's DistributedRobotCommLib with H2 and P2
- R2C::SCAC : Robot2Comm.StartCommunicationAsClient()
- R2C::HTCPC : Robot2Comm.HandleTCPClient()
- MoveLeg : A function in the "ListOfFunctions.h" file of Robot2.

Note that the acronyms used for the functions of *Robot2* are likewise. In the following steps, written flow of the DRL is given:

1. A *DistributedRoboticsLib* object *Robot1* is created with the H1 and PORT1
2. The function handlers of *Robot1* are added by calling *CreateMessageHandler* function
3. In the constructor of *Robot1*, *ListenToAdvertisements* thread started and *Robot1* started to listen advertisements on ADV_PORT.

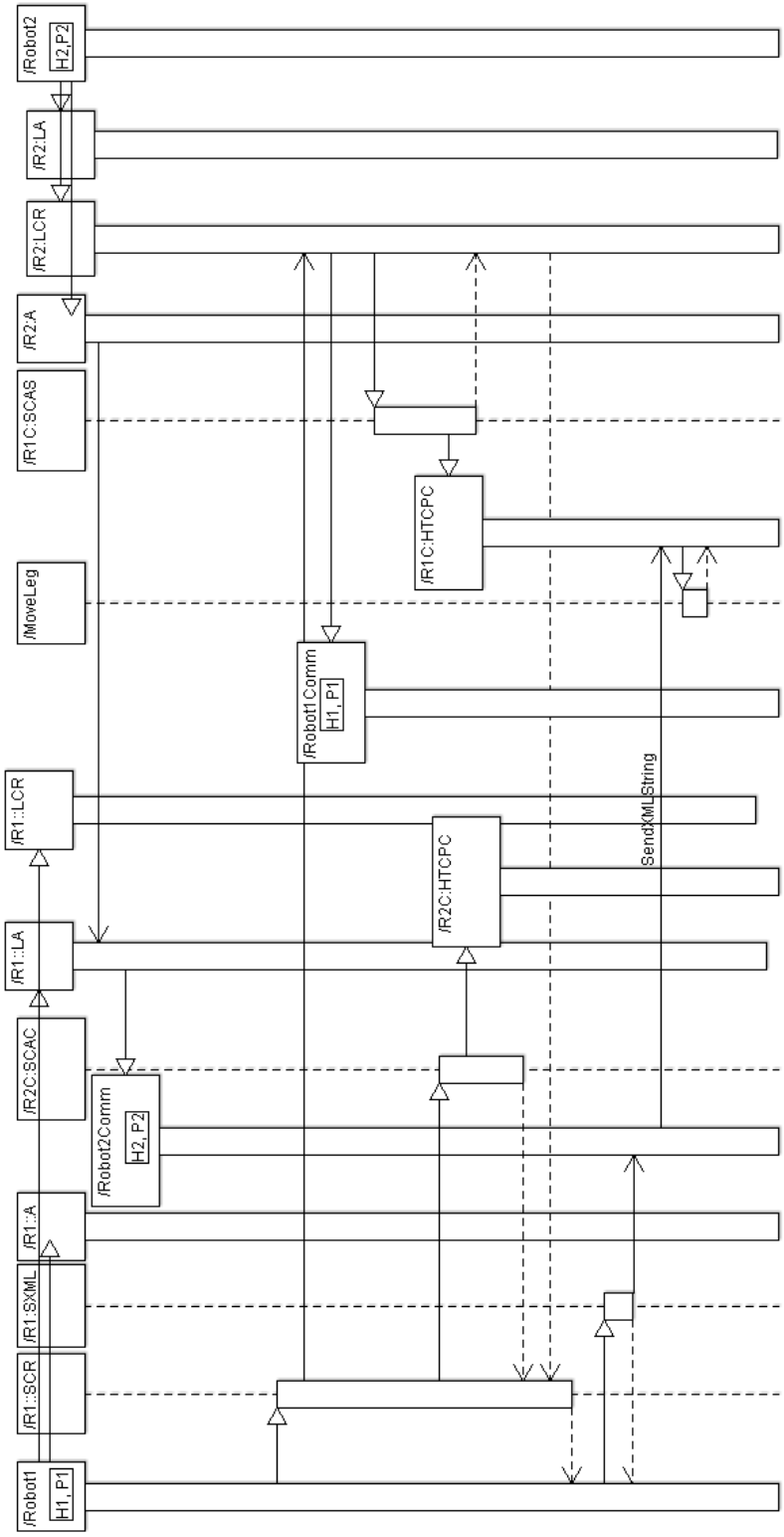

Figure 3-10: UML Sequence Diagram of the DRL Execution.

4. In the constructor of *Robot1*, *ListenConnectionRequests* thread started and *Robot1* started to listen connection requests by creating a server on PORT1.
5. In the constructor of *Robot1*, *Advertise* thread started and advertisements started to be sent each second.
6. A *DistributedRoboticsLib* object *Robot2* is created with the H2 and PORT2
7. The function handlers of *Robot2* are added by calling *CreateMessageHandler* function.
8. In the constructor of *Robot2*, *ListenToAdvertisements* thread started and *Robot2* started to listen advertisements on ADV_PORT.
9. In the constructor of *Robot2*, *ListenConnectionRequests* thread started and *Robot2* started to listen connection requests by creating a server on PORT2.
10. In the constructor of *Robot2*, *Advertise* thread started and advertisements started to be sent each second.
11. *ListenToAdvertisements* thread of *Robot1* detected advertisement of *Robot2* and created a *DistributedRobotCommLib* object *Robot2Comm* with H2 and PORT2.
12. *Robot1* sent a connection request to H2 and started waiting new port number.
13. *ListenConnectionRequests* thread of *Robot2* detected a connection request from H1 and created a *DistributedRobotCommLib* object *Robot1Comm* with H1 and PORT1 since H1 was not on the list. Then it randomized a new port number and sent it to *Robot1*. Then it called *StartCommunicationAsServer* function of the *Robot1Comm*. This function created a server on the new port and started waiting for a connection.
14. After *Robot1* received the new port, it called *StartCommunicationAsClient* function of the *Robot2Comm*. This function created a TCP socket and connected to the TCP server created by *Robot1Comm*. Then, it created a new thread *HandleTCPClient* and started waiting for a message.
15. After TCP server of *Robot2Comm* received a connection, it created a new thread *HandleTCPClient* and started waiting for a message.
16. Now, both parties are ready to send messages to each other.
17. *Robot1* asked *Robot2* to move its leg by sending a proper XML message by calling *SendXMLMessage* function with input H2. *Robot2Comm* is found from the list and *SendXMLString* function is called.
18. Once *HandleTCPClient* thread of *Robot1Comm* received the message, it searched the function list for a match of *MoveLeg*. When it finds the match, callback function is called.
19. *MoveLeg* function in the "ListOfFunctions.h" file of *Robot2* is called and *Robot2* moved its leg to the desired position.

Chapter 4

Experimental Evaluation of Distributed Robotics Library

To test the Distributed Robotics Library (DRL), five Asus Eee PC's are used to represent five different agents(robots) in a network. All the PCs are running 32 bit Microsoft Windows XP Home Edition x86 operating system. The network is created with a wireless router and the PCs are connected to the network via their wireless adapters. In order to build and package the DRL, we used CMake open source build system. CMake is a family of tools designed to build, test and package software. It is used to control the software compilation process using simple platform and compiler independent configuration files. It also generates native makefiles to be used in the compiler environment of the choice[20]. Since we want the DRL to run on both Microsoft Windows and Unix, we used Cygwin to compile and test our library. Cygwin is a Unix-like environment and command-line interface for Microsoft Windows. Cygwin provides native integration of Windows-based applications, data, and other system resources with applications, software tools, and data of the Unix-like environment[21]. The PCs are named as *ALPHA1*, *ALPHA2*, *ALPHA3*, *ALPHA4* and *ALPHA5*. These names are used to identify the robots within the network. Random port numbers are assigned to each robot before running the DRL. A user manual for compiling the library and running applications is given in Appendix A.

Four different test applications are implemented to test the capabilities of the DRL. In Section 4-1, some commands are executed by one robot to test some of the basic functions of the library. In Section 4-2, we implemented an application that synchronizes the clock of the robot by exchanging clock values. In Section 4-3, we show how the DRL provides message relaying in the network and in the last Section, we show the way of handling the relayed message by the sender.

4-1 Testing Basic Functions

Before invoking the basic functions of the library, we create the library object and add function handlers to the object which are working as interfaces between real functionalities of the robot

and the library. The piece of code is as follows:

```
DistributedRoboticsLib robot(hostname,port,info);
VirtualRobot* my_robot=new VirtualRobot(5,10);
robot.CreateMessageHandler("MoveFirstLeg", &MoveFirstLeg, my_robot);
robot.CreateMessageHandler("GetTime",&GetTime,NULL);
robot.CreateMessageHandler("PrintMessage",&PrintMessage,NULL);
```

After creating the library object with a proper hostname and a random port, we create a VirtualRobot object to represent an entity inside the robot. It can be a sensor, an arm, a clock, etc. In our case, we implemented a VirtualRobot class inside the PC which has a leg. Then, we add the message handlers to the robot. As we explained earlier, the first input of the message handler is the tag name to call the function. The second input is the pointer to the function to be handled which is inside "listoffunctions.h" file. The third input is the pointer to the entity inside the robot. By passing this pointer to the interface function in "listoffunctions.h", we enable this interface function to invoke the real function which will do the job. The second and third message handlers are robots own member functions, thus we passed a NULL pointer in the entity input.

After preparing the library, we are ready to test functions of the library. Note that, we run the same application in all PCs, but the following functions will only be called by *ALPHA1*. After running the applications on all PCs, they took care of advertisements automatically and all the PCs are ready to set up a connection between each other.

First function invoked by *ALPHA1* is *GetRobotCommList* function to see the available robots and connection information in the network. Here is the output of this function call:

```
Robot_2@ALPHA1 ~
$ ./console
New Advertisement detected from ALPHA3
New Advertisement detected from ALPHA4
New Advertisement detected from ALPHA5
New Advertisement detected from ALPHA2

List of the robots in the network:
ALPHA3 -Not Connected-
ALPHA4 -Not Connected-
ALPHA5 -Not Connected-
ALPHA2 -Not Connected-
```

As we see from the output, after running the application *ALPHA1* detected advertisements from other robots and we see all the robots in the network, none of them are connected to *ALPHA1*. Then, *ALPHA1* sends connection requests to *ALPHA2* and *ALPHA5* by using *SendConnectionRequest* function and it prompts the list of robots again. The output is as follows:

```
Connection Request sent to ALPHA2 and ALPHA5
```

```
Connection Request Sent To ALPHA2
Client started
Connection Request Sent To ALPHA5
Client started

List of the robots in the network:
ALPHA3 -Not Connected-
ALPHA4 -Not Connected-
ALPHA5 -Connected-
ALPHA2 -Connected-
```

From the output, we understand that both ALPHA2 and ALPHA5 accepted the connection request (which is default in this version of the DRL) and server/client communication processes started in which *ALPHA1* is acting as a client since it sent the connection requests. Then, we prepared the XML messages to be sent and *ALPHA1* sends it by using *SendXMLMessage* function. The output which also includes the XML message sent is as follows:

```
Sending message to ALPHA2
Message sent to ALPHA2
<MoveFirstLeg>
    <X>15</X>
    <Y>20</Y>
</MoveFirstLeg>
```

Here, we sent a message to *ALPHA2*, to call *MoveFirstLeg* in "listoffunction.h" file. Since this function is a "setter" function, we did not add $< HANDLER >$ item to the XML message. *MoveFirstLeg* function simply calls the *MoveFirstLeg* function of the VirtualRobot, which was created at the beginning, with the desired parameters $(X, Y) = (15, 20)$. After calling this function, as we will see in the output of *ALPHA2*, it gets the position of the first leg from VirtualRobot and prompts to the screen. Here is the whole output of *ALPHA2*:

```
Ali@ALPHA2 ~
$ ./console
New Advertisement detected from ALPHA4
New Advertisement detected from ALPHA1
New Advertisement detected from ALPHA3
New Advertisement detected from ALPHA5

Connection Request detected from ALPHA1
Server started

Message received from ALPHA1
<MoveFirstLeg>
    <X>15</X>
    <Y>20</Y>
```

```
</MoveFirstLeg>

First Leg is moved to X:15 Y:20
```

Likewise *ALPHA1*, it also detected advertisements from the other robots. Then, it detected a connection request which was sent by *ALPHA1* and it started server/client communication process as a server. Then it receives the message and calls the function with desired parameters. As we see on the last line, first leg of VirtualRobot is moved to desired position. In figure 4-1, we explain the communication process better with the steps. This figure also clarifies why the VirtualRobot pointer was passed while creating message handler in the first paragraph of this section.

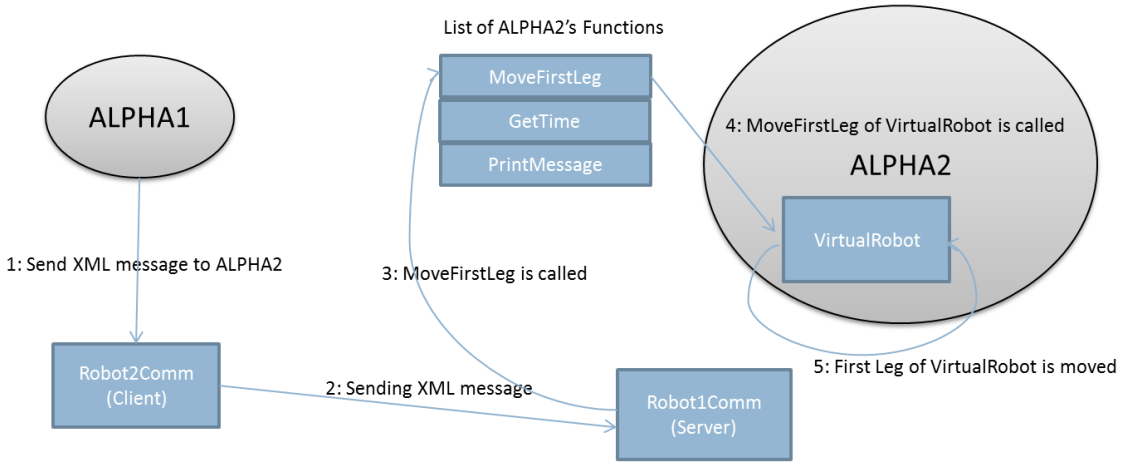

Figure 4-1: XML message sending process from ALPHA1 to ALPHA2.

Now, we test the same *SendXMLMessage* function with another message, which intends to call a "getter" function of *ALPHA5*. First we prepared the XML message and sent it to *ALPHA5*. The output including the XML message is as follows:

```
Sending message to ALPHA5
Message sent to ALPHA5
<GetTime>
    <HANDLER>
        <FUNCTION>PrintMessage</FUNCTION>
    </HANDLER>
</GetTime>

Message received from ALPHA5
<PrintMessage>
    <STRING>Localtime is :  18:12:23</STRING>
```

```
</PrintMessage>

Localtime is :  18:12:23
```

If we look at the XML message in the output, we see that the function tag is sent along with the handler information since *GetTime* is a "getter" function. As we explained in the previous chapter, "getter" functions are sending some data back, and handler function is the function handling this data on the sender side. In this particular example, *ALPHA1* calls GetTime function of *ALPHA5* and it also tells how it will handle the time. On *ALPHA5* side, *GetTime* function reads the clock of the PC, ands prepares XML message to send back to *ALPHA1* with the handler information. As we see, *ALPHA1* receives message form *ALPHA5* which says to call its *PrintMessage* function. As a result, *ALPHA1* asked for the time of *ALPHA5* and told that it will print the time. As we see on the last line, *ALPHA1* prints the time of *ALPHA2*. In figure 4-2, we explain the communication process better with the steps. The rest of the handling functions such as saving received data, are not implemented and left to the library user. The whole output of *ALPHA5* is as follows:

```
Ali@ALPHA5 ~
$ ./console
New Advertisement detected from ALPHA4
New Advertisement detected from ALPHA1
New Advertisement detected from ALPHA3
New Advertisement detected from ALPHA2

Connection Request detected from ALPHA1
Server started

Message received from ALPHA1
<GetTime>
    <HANDLER>
        <FUNCTION>PrintMessage</FUNCTION>
    </HANDLER>
</GetTime>

Message sent to ALPHA1
<PrintMessage>
    <STRING>Localtime is :  18:12:23</STRING>
</PrintMessage>
```

The last tested function is the *StopCommunication* function of the library. In the output, list of robots is given after *ALPHA1* stops communication with *ALPHA2*.

```
Dropping connection with ALPHA2

List of the robots in the network:
ALPHA3 -Not Connected-
```

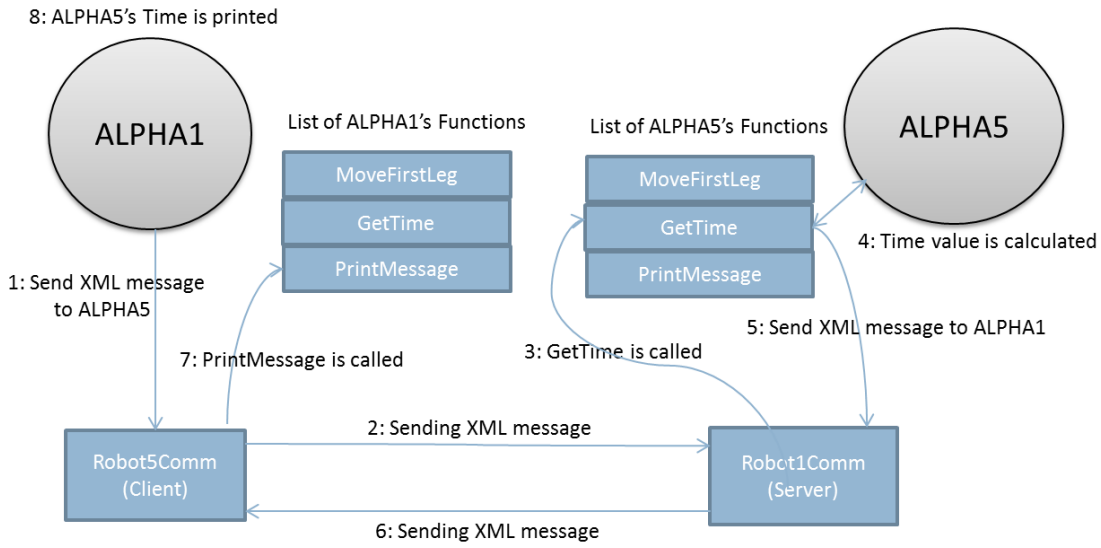

Figure 4-2: An illustration of a "getter" function call.

```
ALPHA4 -Not Connected-
ALPHA5 -Connected-
ALPHA2 -Not Connected-
```

As we see in the output, after stopping the communication, the robot stays in the list. However, server/client communication process is ended by the request.

4-2 Clock Synchronization Application

Clock synchronization is addressing a general problem from computer science and robotics. The problem is that in a Multi Agent System (MAS), although all the clocks are set accurately in the beginning, they may differ in time because of the clock drift. Clock synchronization gives a solution to the agents for converging to a common clock. In this application, we synchronize internal clocks of the robots, by exchanging their clock information.

In this application, we use a predefined network topology which is given in figure 4-3. In order not to change the real clock of the PCs, we added clock functions to the VirtualRobot class. It has a member variable delay and it keeps the track of time by the adding this delay to the real clock. We are keeping the time in milliseconds. Then we added *SynchronizeTime* function which takes the clock of the sender as an input and setting the clock to the arithmetic mean of the sender's clock and its own clock. The XML message to be sent is:

```
<SynchronizeTime>
    <MSEC> 101010010 </MSEC>
```

```
</SynchronizeTime>
```

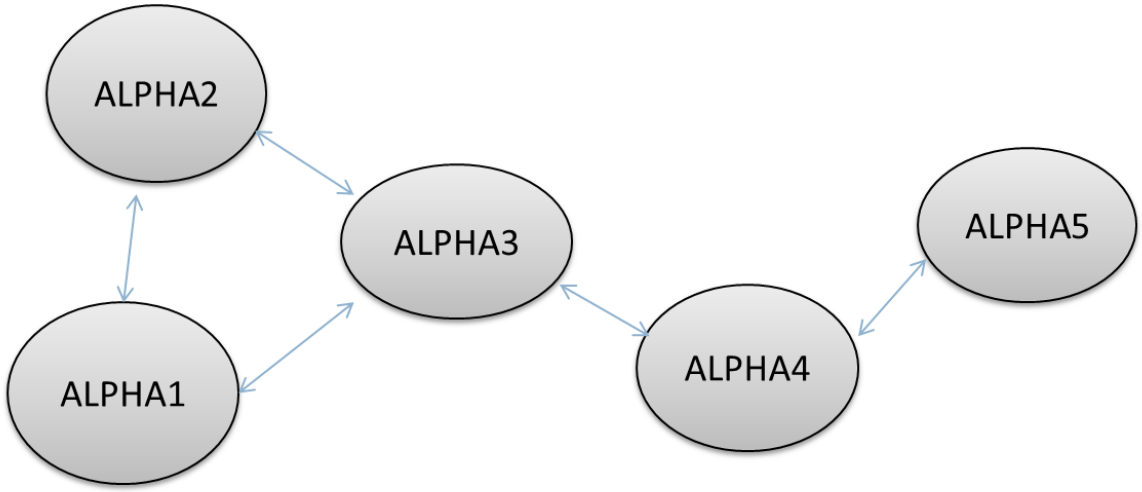

Figure 4-3: Topology of the Network used in Section 2,3 and 4.

Since *SynchronizeTime* function is a "setter" function, we just send the clock of the sender in *MSEC* tag. After establishing the connections, each robot waits for 5 seconds and they start to send the XML message to the robots that they are connected to. They send the same message in each second. At the 15th second, we add a noise of 10 seconds to the clock of *ALPHA5* and at the 25th second, we add the same amount of noise to the clock of *ALPHA3* to see the responses of robots to the noise. In figure 4-4, we see the graph of clocks for 45 seconds. Note that, clocks are shifted towards 0 with respect to the smallest clock.

Before the robots start the clock synchronization(the first 5 seconds), the clocks of the robots increase linearly. After the 5th second, all the clocks converge to the same value. Since *ALPHA5* is connected with only one robot, its settling time is longer than the others. At the 15th second and 25th second, when we add 10 seconds noise to the clocks of *ALPHA5* and *ALPHA3* respectively, there are jumps in corresponding robots' clocks. As we see from the graph, while the clock of *ALPHA3* is settling fast in 2 seconds, it takes about 10 seconds for the clock of *ALPHA5* to settle. As expected, when number of robot's connections increase, the settling time of the clock decrease. In this type of an application, trade off between saving resources and settling time should be taken into account.

4-3 Relaying Messages

Message relaying is essential in a network. Sometimes, in order to save the resources of the network, total number of connections can be limited. In this case, multi hop communication is used between directly unconnected agents and message must be relayed by the intermediate nodes. In Bluetooth scatternets, some nodes are used to connect two different piconets. These intermediate nodes must have message relaying attribute in order to provide communication

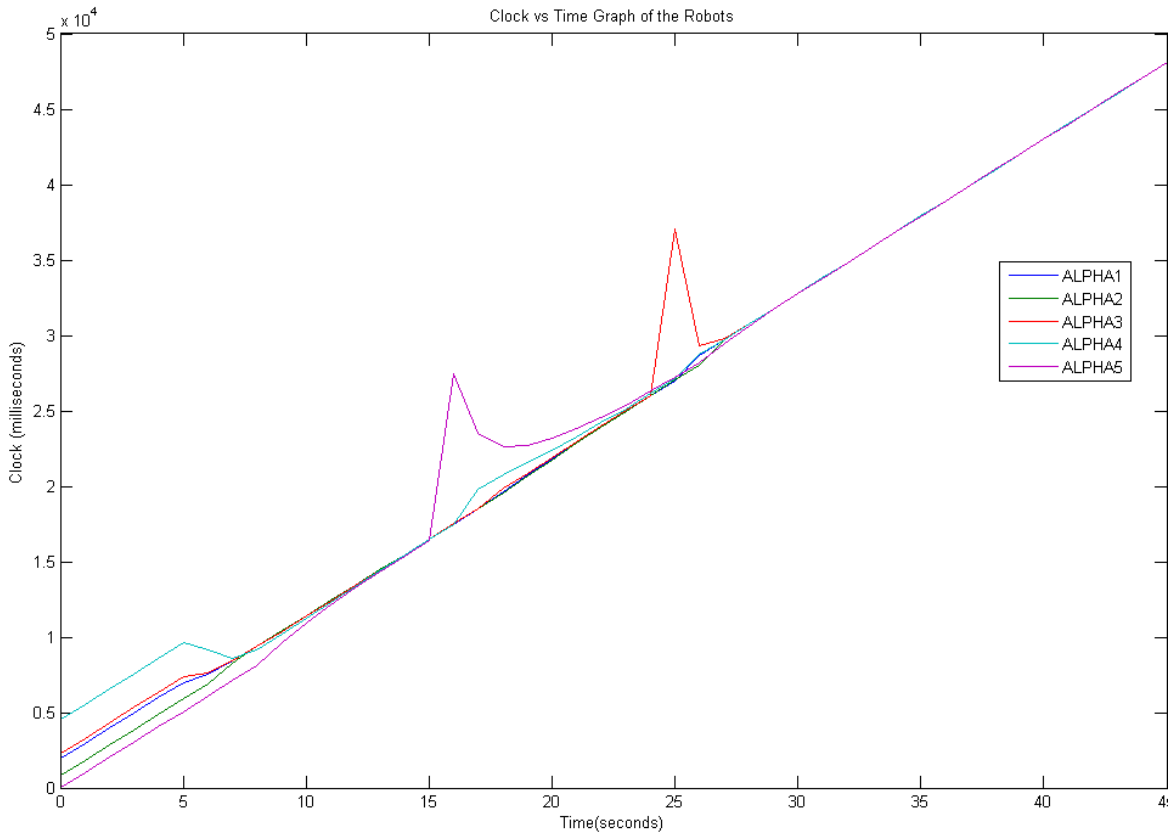

Figure 4-4: Clocks of The Robots In Time.

between the piconets. Although in this version of the library, a robot cannot be connected to two different Wi-Fi networks so cannot be intermediate node between networks, in future protocol-free versions of library, Bluetooth kind of technologies can be used and a robot can be used to transfer messages between networks to increase the range of the communication. In this case, message relaying would also be essential for the intermediate robot.

In this application, we will answer how can ALPHA1 send a message to ALPHA5 without establishing a connection in the network shown in figure 4-3. At first glance, the DRL does not seem to provide any function to relay messages within the network. The reason is that, the DRL is designed to set up connections in the network and enable easy data transfer between two communication entities. From this point of view, message relaying would be a complicated task for DRL. However, DRL allows us to call the functions of any robot by sending a single XML message. If we consider message relaying as a feature of the robot rather than something that DRL provides, then we should achieve message relaying by adding an extra software feature to the robot. This is done by adding a simple relay function to "listoffunctions.h".

To be able to send messages by using multi hop relay, first thing is to determine a path between the sender and the receiver. However, this should be something to be done in user level rather than losing the simplicity of the DRL. In this application, we assume that the path is predefined as $ALPHA1 \rightarrow ALPHA3 \rightarrow ALPHA4 \rightarrow ALPHA5$. These paths can be found by implementing optimal path finding algorithms. As the relaying function, we implemented *SendToOtherRobot* function which is called by an XML message provided with receiver and real function information. For instance, to call *SynchronizeTime* function of *ALPHA4* from *ALPHA1*, the following message should be sent to *ALPHA3*:

```
<SendToOtherRobot>
    <RECEIVER>ALPHA4</RECEIVER>
    <FUNCTION>
        <SynchronizeTime>
            <MSEC> 101010010 </MSEC>
        </SynchronizeTime>
    </FUNCTION>
</SendToOtherRobot>
```

As we see, we are adding two child items, one is receiver and one is the real function to be called. When this message is sent to *ALPHA3*, *SendToOtherRobot* function is called from "listoffunctions.h" as expected. In this function, the file is parsed and the sub document inside the $< FUNCTION >$ item is sent to the receiver. At the receiver side, the function in the sub document is called which is intended. Multi hop relay communication is also possible by using nested *SendToOtherRobot* tags provided with the next receiver information.

After providing relaying function, we tested this function by calling *TurnOffMonitor* function of *ALPHA5* from *ALPHA1*. After setting up the connections shown in figure 4-3, we send the XML message to from *ALPHA1* to *ALPHA3*. The outputs including sent and received messages are as follows:

Output of *ALPHA1*:

```
Message sent to ALPHA3
<SendToOtherRobot>
    <RECEIVER>ALPHA4</RECEIVER>
    <FUNCTION>
        <SendToOtherRobot>
            <RECEIVER>ALPHA5</RECEIVER>
            <FUNCTION>
                <TurnOffMonitor/>
            </FUNCTION>
        </SendToOtherRobot>
    </FUNCTION>
</SendToOtherRobot>
```

Output of *ALPHA3*

```
Message received from ALPHA1
<SendToOtherRobot>
    <RECEIVER>ALPHA4</RECEIVER>
    <FUNCTION>
        <SendToOtherRobot>
            <RECEIVER>ALPHA5</RECEIVER>
            <FUNCTION>
                <TurnOffMonitor/>
            </FUNCTION>
        </SendToOtherRobot>
    </FUNCTION>
</SendToOtherRobot>

Message sent to ALPHA4
<SendToOtherRobot>
    <RECEIVER>ALPHA5</RECEIVER>
    <FUNCTION>
        <TurnOffMonitor/>
    </FUNCTION>
</SendToOtherRobot>
```

Output of *ALPHA4*:

```
Message received from ALPHA3
<SendToOtherRobot>
    <RECEIVER>ALPHA5</RECEIVER>
    <FUNCTION>
        <TurnOffMonitor/>
    </FUNCTION>
```

```
</SendToOtherRobot>

Message sent to ALPHA5
<TurnOffMonitor/>
```

Output of *ALPHA5*:

```
Message received from ALPHA4
<TurnOffMonitor/>

Monitor is turned OFF\
```

As we see from the outputs, we used nested *SendToOtherRobot* functions and inside the inner one, we send the real function. In the output of *ALPHA3*, we see that message is parsed and *function* sub document is sent to the receiver. In the output of *ALPHA4*, we see that message is parsed again and function sub document is sent to receiver ALPHA5. In the output of *ALPHA5*, message received, parsed and *TurnOffMonitor* function that we added to "listoffunctions.h"is called. We also saw visually that, the monitor of *ALPHA5* was turned off.

4-4 Handling Relayed Messages

In the first section of this chapter, we showed the difference of XML messages between calling "setter" and "getter" functions. In order to get some data from another robot, the sender should send the handler function in the XML message. In the previous section, we showed how to relay messages in the network by using the DRL, but still we did not solve the problem of getting data from a robot that is not directly connected. Since we already have the relaying function, all we have to do is to find a way to keep the path information till the receiver since it will need that information to send the data back.

Since we are sending handler information in XML message, and we have the route information before starting, we can add the sender information to the handler. If there is multi hop relaying, then more than one sender can be added to the handler. If we use the same path as the previous section, then the handler item will be as follows:

```
<HANDLER>
    <SENDER>ALPHA3</SENDER>
    <SENDER>ALPHA1</SENDER>
    <FUNCTION>PrintMessage</FUNCTION>
</HANDLER>
```

As we see in the message, the last sender is added as the first sender as a child node. Note that, there is no need to add *ALPHA4* as a sender, because *ALPHA5* will receive the message from *ALPHA4* connection and it will automatically send the reply message back to it. When the final receiver gets the XML message, it parses the message and prepares a reply XML message to be sent to the first sender. As an extension of the library, we added *PrepareReturnMessage* function which takes handler sub document as an input and prepares the return message by using *SendToOtherRobot* tag names. Then, the receiver adds data to the prepared message and sends it back. To test this functionality, we sent *GetTime* function by using the same path. The outputs of the robots including the messages they sent are as follows:

Output of *ALPHA1*

```
Message sent to ALPHA3
<SendToOtherRobot>
    <RECEIVER>ALPHA4</RECEIVER>
    <FUNCTION>
        <SendToOtherRobot>
            <RECEIVER>ALPHA5</RECEIVER>
            <FUNCTION>
                <GetTime>
                    <HANDLER>
                        <SENDER>ALPHA3</SENDER>
                        <SENDER>ALPHA1</SENDER>
                        <FUNCTION>PrintMessage</FUNCTION>
                    </HANDLER>
                </GetTime>
            </FUNCTION>
        </SendToOtherRobot>
    </FUNCTION>
</SendToOtherRobot>

Message received from ALPHA3
<PrintMessage>
    <STRING>Localtime is :  15:14:29</STRING>
</PrintMessage>

Localtime is :  15:14:29
```

Output of *ALPHA3*

```
Message sent to ALPHA4
<SendToOtherRobot>
    <RECEIVER>ALPHA5</RECEIVER>
    <FUNCTION>
        <GetTime>
            <HANDLER>
```

```
                    <SENDER>ALPHA3</SENDER>
                    <SENDER>ALPHA1</SENDER>
                    <FUNCTION>PrintMessage</FUNCTION>
                </HANDLER>
            </GetTime>
        </FUNCTION>
    </SendToOtherRobot>

    Message sent to ALPHA1
    <PrintMessage>
        <STRING>Localtime is :  15:14:29</STRING>
    </PrintMessage>
```

Output of *ALPHA4*

```
    Message sent to ALPHA4
    <SendToOtherRobot>
        <RECEIVER>ALPHA5</RECEIVER>
        <FUNCTION>
            <GetTime>
                <HANDLER>
                    <SENDER>ALPHA3</SENDER>
                    <SENDER>ALPHA1</SENDER>
                    <FUNCTION>PrintMessage</FUNCTION>
                </HANDLER>
            </GetTime>
        </FUNCTION>
    </SendToOtherRobot>

    Message sent to ALPHA1
    <PrintMessage>
        <STRING>Localtime is :  15:14:29</STRING>
    </PrintMessage>
```

Output of *ALPHA5*

```
    Message sent to ALPHA4
    <SendToOtherRobot>
        <RECEIVER>ALPHA3</RECEIVER>
        <FUNCTION>
            <SendToOtherRobot>
                <RECEIVER>ALPHA1</RECEIVER>
                <FUNCTION>
                    <PrintMessage>
```

```
                        <STRING>Localtime is :  15:14:29</STRING>
                    </PrintMessage>
                </FUNCTION>
            </SendToOtherRobot>
        </FUNCTION>
    </SendToOtherRobot>
```

In outputs of *ALPHA3* and *ALPHA4*, we see that the received XML messages are relayed to the receiver. In the output of *ALPHA5*, the received message calls the *GetTime* function and since it is a "getter" function, an XML message is prepared to send the time back to *ALPHA1*. In the output of *ALPHA1*, we see that after getting the information with the handler function *PrintMessage*, it prints the clock of *ALPHA5*.

Since "ListOfFunctions.h" is created by user, a user can use different protocols to relay the messages or to handle the relayed messages. In this section and previous section, we gave our own protocols to handle message relaying and we provided some functions as an extension to the library.

Chapter 5

Conclusions and Future Work

5-1 Summary and Conclusions

The goal of this graduation project was to develop a communication library to be used by distributed robotics networks. The main requirements of the library were portability to different operating systems, to provide automatic identification in the network, to provide fast and simple ways of communication, to enable parallel communications, to provide common structure for handling messages and to be open to improvements. While designing the Distributed Robotics Library (DRL), these requirements were taken into consideration and they were tried to be provided as much as possible.

A communication library design requires set of protocols from physical connection to the user. These set of protocols are layered and structured in the 7 Layer OSI Reference Model. This model is quite abstract and any communication system is modeled around this model. In designing of the DRL, this model is utilized.

Communication libraries that are used for distributed robotics are generally application specific and in order to get the best performance, all protocols are set to be suitable to the applications. On the other side, the DRL is designed independent of any application. For this reason, in lower layers of the model, it uses general protocols. For physical and data link layers, it is designed to use Wi-Fi wireless standard. Hence, the agents should be Wi-Fi certified in order to use the DRL. The network layer also uses the most prominent protocol which is IPv4 protocol.

There are three main processes in the DRL which are advertisement process, connection request process, and server/client communication process. These processes have different set of rules which means the DRL uses different protocols for upper layers. In network layer, both TCP and UDP are used. UDP is used by advertisement process for broadcasting and TCP is used by the other processes for one-to-one communication. The DRL is implemented in C++ Programming Language. In order to use TCP and UDP protocols with C++, "Practical C++ Sockets" interface class is utilized. For representation of messages, XML encoding is

used. XML is chosen because it is generally accepted, widely used and easy to understand. CMarkup XML parser class is utilized for encoding and decoding of messages.

The DRL is using multi-threading in order to run different processes independent of each other. This parallel way of execution, increases response time of the robots and reduces the propagation delay of messages trading off with the system resources. In this version of the library, we assumed that each robot has enough resources to handle the communications and the library is designed for maximum response time and minimum propagation delay.

For testing the DRL, five Asus Eee PC's are used as robots. Although they do not have real actuation capabilities, they have internal data to exchange and some attributes to set which are enough for testing. In order to use the DRL, one should add "listoffunctions" to the library which plays interface role between robot's real functions and common function representation of the library. The functions in "listoffunctions.h" file represents the attributes of the robot to be used in the network. We added some functions for testing and in our routine function tests, all the functions worked properly.

As we said, the DRL's perception of the robot is limited to "listoffunctions.h". Hence, it is possible to add some abstract attributes to the robot by adding functions to this list. In Chapter 4, we add one abstract attribute which is relaying messages. Since multi hop communication is important to balance propagation delay and network resources, message relaying is vital for a robot. By adding relay function and by calling this function in a proper way, the DRL provides message relaying in the network.

In conclusion, the DRL is a simple communication library, easy and fast to use for simple applications. However, the significance of the DRL is that it is open to application specific improvements and it provides a design path for an advanced communication library.

5-2 Future Work

- The maximum message size that the DRL can send is limited with the limit in the Berkeley sockets application programming interface (API) which is used by the "Practical C++ Sockets". Although, this limit is not small(2 KB for TCP, 30 KB for UDP) to send simple messages, it may be small for sending different multimedia contents. In order to send large bulks of data, a protocol can be introduced and a new function can be added to the library.

- In *DistributedRoboticsLib*, there is a pointer "info" which is passed to the object by the user. This pointer comprised of some information about the robot. In this version of the library, we pass empty string for the information of the robot since we assume that we know the other robot's functions. As an improvement, this information can be filled with the attributes of the robot and this information can be exchanged once in a while to be aware of the capabilities of the network.

- Across the planet, communication is supplied by TCP/IP independent of the physical connection such as Ethernet, wireless, etc. In the current version of the library, we use Wi-Fi and IPv4 standards which restrict the library. For local networking, services of TCP/IP protocol are more than necessary and it requires a strong infrastructure. In Chapter 2, we explained Zero Configuration Networking which gives a guide to design

local networking protocols. Similar set of protocols can be designed for lower layers of the DRL to make it more independent and more suitable for the intended applications.

- For supervision of the network, a special node can be introduced. In order to enable this node to control connectivity of the network, simple special functions can be added to the "listoffunctions" of the robots by default. These functions will allow the supervisor node to use the library functions of other robots and the supervisor node will be able to establish a communication between two robots.

Appendix A

User Manual

A-1 Creating your own application

The Distributed Robotics Library (DRL) is packaged in a single file called "thesis_library" and provided as a zip file called "thesis_library.zip". After installing the "thesis_library.zip" on your system, you need to follow these steps:

1. Unzip the file to your system.
2. Go to \thesis_library\source\applications\SimpleRobotLibrary folder.
3. Add header and cpp files of all the entities in your robot and add interface functions' declarations to "ListOfFunctions.h" and definitions to "ListOfFunctions.cpp". Note that, there are some useful functions already provided with the library.
4. Make the proper changes to the "CMakeLists.txt" file of the folder. Now your robot is ready to use the library.
5. Go to \thesis_library\source\applications folder.
6. Add a folder for your application and make proper changes to the "CMakeLists.txt".
7. Add your application's header and cpp files and create a "CMakeLists.txt" file for your application.
8. It is possible to add multiple application in the same way.

A-2 Compiling the library and applications

After adding the robot files and application files go back to the home folder of library (~ thesis_library) and follow these steps:

1. Run prepare file in order to create folders and makefiles. If your system is mission any of the packages needed, you will get warnings to install them.
2. Run “make” command in order to create binary files. Note that, different makefiles are created for each folder to compile them separately.
3. Go to \thesis_library\binaries\applications folder where you can find the executable file.
4. You can copy and use the executable on different systems.
5. In order to create documentation files, run “make doc” command.
6. In \thesis_library\documentation folder, you can find html and latex documentation files of the library.

A-2-1 Compilation Tips for Windows Users

If you are using a Windows system, it is strongly recommended to install “cygwin” cross-compiler in order to use “CMake” and “Doxygen” properly. Then, it is possible to compile the library by following the same steps explained above. Note that the executable files are windows executable now and can be run on Windows systems by either adding location of the “cygwin1.dll” file to the system path, or adding the file to the same folder with the executable.

Bibliography

[1] H. Benítez-Pérez and F. García-Nocetti, *Reconfigurable Distributed Control*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[2] J. Galdun, L. Takac, J. Ligus, J. Thiriet, and J. Sarnovsky, "Distributed control systems reliability: Consideration of multi-agent behavior," in *6th International Symposium on Applied Machine Intelligence and Informatics, 2008. SAMI 2008.*, pp. 157 –162, 2008.

[3] J. Aguilar, M. Cerrada, G. Mousalli, F. Rivas, and F. Hidrobo, "A multiagent model for intelligent distributed control systems," in *Knowledge-Based Intelligent Information and Engineering Systems* (R. Khosla, R. J. Howlett, and L. C. Jain, eds.), vol. 3681 of *Lecture Notes in Computer Science*, pp. 191–197, Springer, 2005.

[4] F. Ruini, "Communication and distributed control in multi-agent systems: Preliminary model of micro-unmanned aerial vehicle (mav) swarms," 2008.

[5] M. Koes, I. Nourbakhsh, and K. Sycara, "Constraint optimization coordination architecture for search and rescue robotics," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA) 2006*, pp. 3977–3982, May 2006.

[6] V. Trianni and M. Dorigo, "Self-organisation and communication in groups of simulated and physical robots," *Biological Cybernetics*, vol. 95, pp. 213–231, August 2006.

[7] V. Julián and C. Carrascosa, "Physical agents," in *Issues in Multi-Agent Systems*, Whitestein Series in Software Agent Technologies and Autonomic Computing, pp. 117–143, Birkhäuser Basel, 2008.

[8] F. Bullo, J. Cortes, and S. Martinez, *Distributed Control of Robotic Networks: A Mathematical Approach to Motion Coordination Algorithms*. Princeton, NJ, USA: Princeton University Press, 2009.

[9] R. Hekmat, *Ad-hoc Networks: Fundamental Properties and Network Topologies*. Springer, 2006.

[10] D. Wetteroth, *OSI Reference Model for Telecommunications*. McGraw-Hill Professional, 2001.

[11] ITU-T Recommendation X.200, "Information technology — open systems interconnection — basic reference model: The basic model," 1994.

[12] A. S. Tanenbaum, *Computer networks: 2nd edition*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[13] W. R. Stevens, *TCP/IP illustrated (vol. 1): the protocols*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1993.

[14] H. Labiod, A. Hossam, and C. D. Santis, *Wi-Fi, Bluetooth, Zigbee and WiMax*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[15] P. Kinney, "Zigbee technology: Wireless control that simply works." www.zigbee.org/resources, Oct. 2003. Whitepaper.

[16] D. Steinberg and S. Cheshire, *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, Inc., 2005.

[17] S. J. Donahoo and K. L. Calvert, *TCP/IP Sockets in C: Practical Guide for Programmers*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2002.

[18] E. T. Ray, *Learning XML*. O'Reilly Media, 2001.

[19] First Objective Software, Inc., "Cmarkup c++ xml parser." http://www.firstobject.com/, 1999–2009.

[20] A. Cedilnik, B. Hoffman, B. King, and A. N. Ken Martin, "Cmake." http://www.cmake.org/, 2000–2010.

[21] Cygnus Solutions, "Cygwin." http://www.cygwin.com/, 1995–2010.

Glossary

List of Acronyms

DRL	Distributed Robotics Library
MAS	Multi Agent System
OSI	Open Systems Interconnection
DCS	Distributed Control System
DRS	Distributed Robotics System
DCSC	Delft Center for Systems and Control
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
ASCII	American Standard Code for Information Interchange
XML	Extensible Markup Language
ISO	International Organization for Standardization
FHSS	Frequency Hopping Spread Spectrum
DSSS	Direct Sequence Spread Spectrum
OFDM	Orthogonal Frequency Division Multiplexing
HR-DSSS	High Rate Direct Sequence Spread Spectrum
WLAN	Wireless Local Area Network
WPAN	Wireless Personal Area Network
DLL	Data Link Layer
LLC	Logical Link Control

MAC Media Access Control

VLAN Virtual Local Area Network

CSMA/CD Carrier Sense Multiple Access With Collision Detection

LAN Local Area Network

CSMA Carrier Sense Multiple Access

CSMA/CA Carrier Sense Multiple Access With Collision Avoidance