Politecnico di Torino

Collegio di Ingegneria Informatica, del Cinema e Meccatronica

MSc in Mechatronic Engineering

ROS over the Internet: developing a VPN-based ROS network to enable remote communication using a 3D printed robot hand

Supervisors

Prof. Paolo Prinetto

Author

Pecone Cesare

Supervisors at HotBlack Robotics:
Dr. Ludovico Orlando Russo
Dr. Gabriele Ermacora

March 2018

Abstract

Acknowledgments

Contents

List of Tables

Listings

Chapter 1

Introduction

... in this part will be argumented an introduction to this Master Thesis, presenting a brief discussion of the main topics addressed during the project development: Robotics, Cloud Robotics, Networks, etc..

1.1 This Master Thesis

This Master Thesis has been developed from October 2017 to March 2018 in the Robotic Research Group (RRG) laboratory of Politecnico di Torino, under the supervision of Prof. Paolo Prinetto. In addition, since a collaboration with the HotBlack Robotics group was established, also Dott. Ludovico Russo and Dott. Gabriele Ermacora helped in the project. This Thesis represents the Final Project of the Master Degree in Mechatronic Engineering, and it covers several topics studied within this degree, together with the author's personal background and experience. Among the subjects associated with this work, the main ones are: Robotics, Networks, Operative Systems, Software Design and Development, Controls, Computer Science...

1.1.1 Main Objectives and Results

The main objective of This Master Thesis is the study and development of a network system allowing to use a 3D printed hand over the Internet. In order to reach such goal, several topics have been studied and discussed during the project. In particular, the following paradigms have been encountered: Cloud Computing, Robotics and the application of Cloud Computing features to Robotics, to achieve what is known as Cloud Robotics.

The strategy implemented can be divided in the following steps:

1. First...
2. Second...
3. Third...

The project has led to many useful results... The main Thesis outcomes can be summarized as follows:

- The VPN Multi-Machine ROS network has showed good results...
- ...

- ...
- ...

1.1.2 Outline

This thesis is organized in six Chapters described below.

The first chapter (this Chapter) provides introduction to this Master Thesis, presenting a brief discussion of the main topics addressed during the project development: Robotics, Cloud Robotics,Networks, etc...

The second chapter contains a general overview of Robotics, Cloud Robotics and Robot Operating System (ROS). The aim is to provide a theoretical review of the main concepts to understand the developed work...

The third chapter The main features of the Parloma Project are presented, together with the Author contributions to some Parloma open issues...

The fourth chapter The VPN Multi-Machine ROS network developed is explained in details. Not only the theoretical concepts behind it are presented, but also a step-by-step description of the implementation process is provided.

The fifth chapter Two actual case studies making usage of the VPN Multi-Machine ROS network developed are addressed. Two configurations will be tested: single client/server and multiple client/server.

The sixth and last chapter A conclusion to this Master Thesis is presented, including open issues, possible future developments and achieved results.

Chapter 2

Background

This chapter contains a theoretical overview of Robotics, Cloud Robotics, Robot Operating System (ROS), Virtual Private Network and Operating-System-Level Virtualization. The Chapter aim is to provide the main theoretical concepts indispensable to understand the work done during this Thesis. Moreover, some examples and clarifications are presented in order to simplify some notions.

2.1 Robotics

Robotics is a branch of engineering and science dealing with the design, construction, operation and use of robots: a machine capable of automatically carrying out a complex series of actions automatically. Besides, it includes also the study of computer systems for robots control, sensory feedback and information processing. Clearly, robotics is an interdisciplinary science, since it groups several engineering fields, together with biology, psychology and philosophy, that address more deep topics associated with robotics. These technologies have been firstly studied and developed in order to substitute humans in different situations as: dangerous environments, manufacturing processes and others. However, in the last years, the trend is focusing on service robotics, that is a new form of robotics related to every day life.

Among all the different robotics applications, the following are the ones of main interests:

- **Industrial Robots** are manipulators designed to move materials, parts and tools performing different tasks in manufacturing and production environments.
- **Service Robots** are planned to assist humans in every day life. They have a wide range of applications: from cleaning the floor, to assist elderly people. Usually they are fully autonomous and designed to operate in a domestic environment in contact with people. For example, the following are part of Service Robotics:
  - **Edu-tainments Robots** are used to educate children through entertainment. They can be used to assist a teacher or to autonomously teach some basic concepts through fun.
  - **Agricultural Robots**, or agbots, are deployed for agricultural purposes. They cover different tasks in factory settings; the main areas are: weed control, cloud seeding,planting seeds, harvesting, environmental monitoring and soil analysis.

- **Medical Robots** include robots used in medical sciences. Among the different uses, the most popular is surgical robots, where a robot is performing a low invasive operation while it is remotely controlled by an expert.
- **Exploration and Scientific Robots**, also known as rovers, are designed to explore areas difficult to be reached by humans. Besides, they are also able to collect and analyze samples or to send real-time data to a remote station. A well known example is the Mars rover that propelled itself across the Mars planet surface.

2.1.1 Teleoperated Robots

Teleoperation systems were the predecessors of networked robots. They started as remotely controlled devices and then, thanks to the evolution of Internet and related technologies, they became an integration of robots, human, agents, off-board sensors, database and cloud over the globe. The first functioning networked telerobot was built in 1994, under the name of "Mercury Project". It consisted of an IBM industrial robot arm with a digital camera and allowed remote users using the air nozzle to excavate for buried artifacts in a sand container [**RobGold**]. Even if most of the networked telerobotic devices consist of a single human operator and a single robot; according to Chong et al. [**Chong2000**], there are four kinds of disposition: Single Operator Single Robot SOSR, Single Operator Multiple Robot SOMR, Multiple Operator Single Robot MOSR and Multiple Operator Multiple Robot MOMR. This extended networked connectivity greatly enhances the potentiality of networked robotics, and allows the employment of techniques as crowd sourcing and collaborative control.

There are several degrees of autonomy for networked robots, the two extreme cases are the following:

- **Teleoperated**, where human supervisors send commands and receive feedback through the network. These are just remotely controlled robots that execute what they are asked for.
- **Autonomous**, where robots and sensors exchange data through the network. In these systems, the sensor network extends the sensing range of the robots. This time a remote user is not needed.

All the different levels of networked robots autonomy are illustrated in Figure 2.1
Networked robots play a great role in allowing people to interact with remote environment. For example, in videoconferencing applications, with respect to a normal videoconferencing system, networked robot can provide far more interactivity. In fact, the physical robot not only represents the remote person, but also transmits multi-modal feedback to the remotely connected person. This technology is referred to as **Telepresence** in literature.

Telepresence

Telepresence refers to a set of technologies allowing a person to feel as if they were present, to give the appearance of being present, or to have an effect, via telerobotics at a place different than the user location. The most popular application of telepresence is videoconferencing which consists streaming audio and video images between two remotely locate users, famous example of software oriented to telepresence videoconferencing is Skype. The benefits adopting this technology are several: the travel costs are cut, the employees work/life balance is safeguarded and their productivity is enhanced.

Additionally, users may be given the ability to perform tasks in the remote location. Manipulation can be useful for some applications and it can be implemented by a robot in the remote

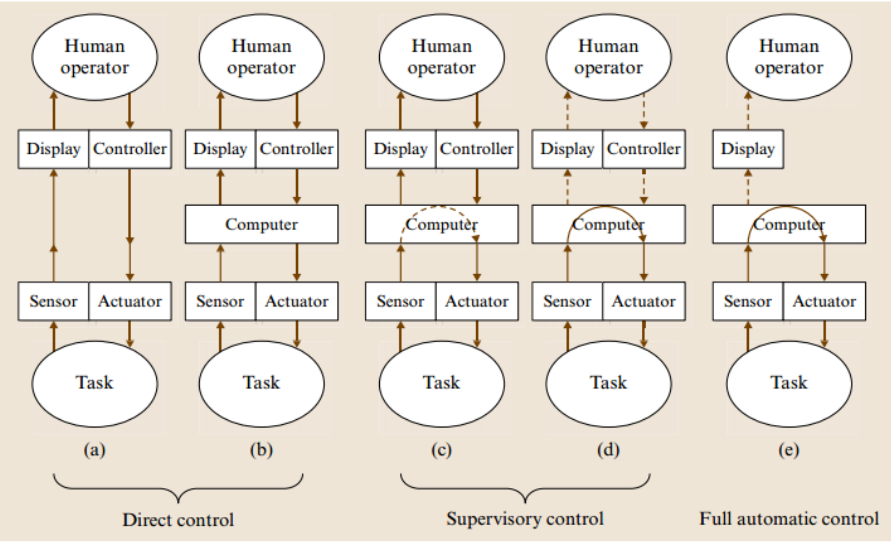

Figure 2.1: A spectrum of teleoperation control modes. At the far left would be a mechanical linkage where the human directly operates the robot from another room through sliding mechanical bars, and on far right would be a system where the human role is limited to observation/monitoring.

location that is copying the movements of the users. Another useful ability is the freedom of movement, that improves significantly the experience of the remote user connected to the screen. Both these two objectives can be achieved by means of a teleoperated robot that follows remote commands while performing the audio/video streaming.
These tasks refer to a new robotics field:
Mobile Robotic Telepresence (MRP) or **Telerobotics**. According to ..., Mobile Robotic Telepresence can be defined as a system that incorporates video conferencing equipment onto mobile robot devices which can be steered and controlled from remote location. These products, which were born to promote social interaction between people, are now becoming popular also in other application domains such as health care environments, independent living for the elderly and office contexts. With respect to traditional videoconferencing, MRP guarantees a deeper social interaction.

Terminology

Before proceeding with some case studies, some specific terms must be defined.

1. **Mobile Robotics Telepresence (MRP)** are systems characterized by a videoconferencing structure mounted on a mobile robotic base. The system allows a remote user to move around in the robot's environment. Its primary objective is to promote social interactions between humans. The MRP subsists of both the physical robot (sensors and actuators) and the interface used to pilot the robot. A typical MRP unit is equipped with: LCD screen, web camera, microphone, speakers and a robotic infrastructure.

2. **Pilot user** is a person who remotely connects to the robot via computer interface. The pilot, embodied into the MRP system, is able to move and interact with other people.

3. **Local user** is the user that stays near the robot. Local users are free to move around while interacting with the pilot user who is visiting them by means of the robot.

4. **Local environment** is the environment in which the robot and the user are located.

Application Areas

The most prolific application areas will be illustrated in this Subsection. In the literature, three particular sectors are pointed out: office environments, health care and elderly assistance (both part of Telemedicine applications). Besides, there are also some works in school environment.

- **MRP Systems for Office Environments.** To mitigate the spatial distance between teams cooperating together, a set of MRP systems have been tested in office environments. They allow remote colleagues to visit the local coworkers, participating in formal as well as informal meetings. Moreover, they provide interactive and expressive channels to allow informal interaction. This kind of interaction is a crucial aspect for a pleasant communication and it is usually brief and not scheduled.

- **MRP Systems for Health Care.** Several tests and evaluations have been performed in the health care sector. These studies, which are still ongoing, show amazing outcomes. Not only the length of permanence due to minor invasive surgery or intensive care units is decreased; but also the response times during emergency situations is reduced.

- **MRP Systems for Elderly and Aging in Place.** Since the world elderly population trend is continuously increasing, aging in place assistance is a huge market for MRP systems. In this scenario they could serve several different tasks: health surveillance, social interaction and safeguarding. For instance, MRP systems can be equipped with medical devices in order to provide data to remote professionals or they could promote social interaction through video calls with the user relatives.

- **MRP Systems for School.** Some MRP systems have been developed to help children as well. The objective is to allow them attending school when their are unable due to hospital recoveries or long term illness at home. The studies have proved that ill students could actively participate at classes as their mates, maintain concentration and communicate successfully.

- **General MRP Systems.** Besides specific applications, nowadays the trend seems to move in building cheap systems consisting of standard hardware and software for robotics (for example ROS) and video communication. This has triggered several devices that have not a specific application domain.

The last area is exactly where the project developed in this thesis can be placed. As will be explained accurately later, the telerobotic system conceived is not limited to a particular context, but it is a more general and flexible application. However, it is useful to provide a general overview of the available technologies.

2.2 Cloud Robotics

The term "cloud-enabled robotics" was used for the first time by James Kuffner at the IEEE RAS Intl. Conference on Humanoid Robotics in 2010. He wanted to highlight the potentialities of distributed networks combined with robotics, in order to improve several aspects of it.

After that, several definitions have been provided, each one focusing on a different aspect of Cloud Robotics, demonstrating the diversification of approaches within the community. Ken Goldberg, for example, accentuates the possibility of a new form of collective robot intelligence thanks to learning and sharing, resulting from the dramatic improvement of computation capacity, memory and programming capability: “Humans as a species are getting smarter because we are able to share information much quicker and build on innovations faster, robots have that potential as well”. Also Steve Cousins strongly believes that the most important opportunity offered by Cloud Robotics is the possibility to share information among robots and letting them communicate through the cloud, since "No robot is an island".

However, one of the most comprehensive definition was provided by Mario Tremblay, who said: "Cloud robotics happens when we connect robots to the Internet and then, by doing so, robots become augmented with more capacity and intelligence. The cloud lets them communicate with other machines and serve their human operators better. Connected robots equal augmented robots. By collaborating with other machines and humans, robots transcend their physical limitations and become more useful and capable, since they can delegate parts of their tasks to more suitable parties".

In more technical words, Cloud Robotics is a field of robotics that makes usage of cloud features such as Cloud Computing, Cloud Storage and other Internet characteristics to extend the benefits of converged infrastructures and shared services to the robotics world. Powerful computation, storage and communication resources are therefore at the service of robots, processing and sharing the information they collect. This disruptive vision of robotics makes it possible to build lightweight, low cost and smarter robots whose computing center is moved into the cloud.

Based on the deployment models presented before, a new form appears: Robotic as a Service (RaaS). As a general understanding, RaaS can be defined as a cloud computing unit that facilitates the seamless integration of robot and embedded devices into Web and cloud computing environment. It follows the SOA (Service Oriented Architecture) and has the following characteristics:

- A Raas unit is a service provider: it has loaded pre-prepared services that can be removed, modified or shared with other robots.
- A Raas unit is a service client: new applications can be deployed on the robot based on the services available.
- A Raas unit is a service broker: a client can navigate through the different services available and organize them in hierarchy classes.

A basic Cloud Robotics architecture can be seen as made of two complementary tiers:

1. **Machine-to-machine (M2M).** The robot are grouped together to form a collaborative computing entity, which leads to several advantages. First, as a result of the collaboration, the computing power is dramatically increased. Then, information can be shared for collaborative decision making. Finally, it extends the communication range of the cloud architecture to robots that are out of it.
2. **Machine-to-cloud (M2C).** On this level, the cloud offers computation and storage resources that are allocated dynamically for real-time demand. In this way, all the heavy computations can be moved from robots to the cloud, together with the large amount of data collected by robots.

Among all the different cloud-based robotics applications, the one of most interest for the aim of this thesis is the Cloud Network (CN). Cloud Network focuses on the network as infrastructure and can be divided into three groups:

- **Teleoperated Robots.** Those robots are remotely controlled by a human operator, and the goal is to achieve telepresence. This field has wide applications in disaster relief, remote exploration, telerobotic mining and surgery.
- **Multi-robot systems.** They are systems where robots are capable to cooperate without human intervention.
- **Sensor arrays.** They can be considered as a form of cloud systems if robots are included in the architecture, as recipients of information from remote sensors through M2M communication.

2.2.1 Benefits

Once that the general concepts related to Cloud Robotics have been addressed and some advantages have been briefly mentioned, it is time to focus on the benefits that the Cloud brings to robotics.

- **Big Data.** The Cloud Storage allows robots to manage Big Data which could have not been maintained on on-board memory. For Big Data are meant extremely large data sets that might include images, videos, maps and so on. There are several examples of applications where Big Data has played a fundamental role together with robotics; among all of them, one of the most interesting is proposed by Kehoe. Grasping objects is a basic tasks addressed by most of the robots, from the industrial robotics to the service robotics. However, it is not so easy to implement all the procedures that start from the object observation to its correct grasping; especially when the robot deals with a never-seen-before object. To this purpose, Kehoe proposed a system that integrates Google Goggles recognition engine with a sampling based algorithm to recognize objects and suggest grasping approaches. The robot takes a photo of the object under analysis and sends it to the server; here a trained algorithm compares the photo from the robot to a large database of CAD models. Once the object is identified, the server sends back the recommended grasping path to the robot. Finally the robot performs the grasping selecting the best approach among those suggested and sends back the feedback to the server that is updated.
- **Cloud Computing.** Having the resources to perform massively-parallel computation is crucial in robotics. Cloud Computing power has two fundamental consequences for robots:
  - First, equip a robot with an Internet connection to use external Cloud Computing power, instead of mounting huge processors allows to cut off costs dramatically.
  - Second, being the heavy hardware moved from the robot to the cloud, robots can perform the same tasks as before, but being lighter and smaller. This second consequence must not be underestimated since most of the applications require light robots and smaller ones.this way, also cheap robots are able to perform heavy computing tasks as problems related kinematics, dynamics, motion planning, decision making under uncertainties (Monte Carlo analysis) and other.

  However, this is possible under the firm hypothesis that network latency is negligible (for on-demand tasks) and that quality of service (QoS) is high.
- **Collective Robot Learning.** The Cloud gives the possibility to share information between robots and to send feedback from the robot back to the servers. This opportunity provides two important advantages:
  - The cloud database takes advantage not only of the information coming from one robots, but from a collectivity of units that acquire useful data.

  - Once the data are collected, fusion and learning methods can enrich and elaborate available information.

  An example of Collective Learning is provided by vehicle route planner such as Google MapsWaze. They start from a set of information retrieved by Satellites, Maps, Imagery and so on,order to have always updated information. Then, we can imagine different users as a community of robots that are continuously providing data on traffic congestions or favorite routes. These data are elaborated through a machine learning algorithm that uses them to choose the best routecase of traffic jams and then provides it to users. This is exactly an example of fusion between existing data and data retrieved from users.

- **Crowdsourcing.** There are some tasks as image labeling, face recognition or learning associations between object labels and locations, where human skill and experience are essential. The Cloud gives the possibility to assist robots from remote, in order to train them for particular tasks. In this scenario, robots can perform their work repeatedly and then understand when they are stuck in something or unable to perform some functions. In such cases, they are able to ask for assistance, and a remote operator helps them by training them properly.

2.2.2 Challenges

Apart from the benefits deriving from the Cloud, there are obviously also some challenges that need to be addressed and yet overcome.

1. Firstly, as mentioned before for Cloud Computing, one of the most relevant problem concerns security and privacy. The privacy issue arises when robots share sensed information to the cloud regarding imagery, videos or others related to private homes or business. These information are accessible also from others and therefore they should be kept protected in some way. On the other hand, the security problem concerns possible hackings to cloud connected robots in order to cause damages to people or to physical or virtual resources. An example of cloud robotics hacking has been performed by the Cockrell School of Engineering where the professor Todd Humphreys, together with his students, proved that spoofing a UAV is practically and technically feasible. This demonstration raised concern and pressed the FAA (Federal Aviation Administration [USA]) to strengthen the regulations of th UAV, requiring to acquire a license in order to guide a UAV in US air space.

2. Another challenge that is strictly connected to Cloud Robotics and, in general, Cloud Computing, is the network latency and the QoS. Even if Internet has grown extraordinarily in the last years, several are the reasons that can cause a loss of speed or an increase of latency. The real challenge here is to be able to design algorithms that are acting accordingly to the QoS of the network, in order to degrade slowly when the quality of the connection is poor. Otherwise, alternative communication channels could be designed for the only purpose connecting the robot to the cloud. However, this problem shows up mainly for real-time applications, where hard real-time computations are required. Whereas, when there is no need of fast response time, it does not represent a huge limitation. For instance, in path planning applications, there is no need of real-time response and therefore the network latency is not a big deal.

3. Finally, a big issue related to Cloud Robotics and, moreover, to Big Data, is to recognize dirty noise and unreliable information. Having confidence in wrong information could cause big damages to both robots and data that are treated, that is why it is crucial to be able to recover only reliable data from all the set of information available. Usually, to deal with noisy information, Data Mining algorithms are used.

2.2.3 Requirements

To complete this theoretical part on Cloud Robotics, the minimum system requirements needed to run a Cloud Robotics architecture are listed.

- Firstly, as base requirement, an Internet connection is needed. It must be as stable as possible and with a good QoS. Obviously, the connection that seems more appropriate is the Local Area Network (LAN). However, being it based on physical wires, it could be not suitable for certain applications. On the other hand, the Wireless Local Area Network (WLAN) is more convenient, but it suffers more from instability.
- Once the connection is established, a computing unit is necessary to perform all the off-loaded computations and to manage the cloud communication.
- Then, a proper software is needed to form a networking allowing the robot to communicate with the cloud and with other units. Lately, the Robot Operating System (ROS) has been widely adopted and will be described in details.
- Finally, a platform able to implement the required architecture is indispensable to complete the set up of the environment.

2.3 ROS: Robot Operating System

ROS stands for Robot Operating System, even if it is not actually an operating system, it is more a middleware that lifts between the actual operating system and software programs. It is a framework that is widely used in robotics.

As described in the official ROS wikipedia[1]: *ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.*

ROS is composed by four different elements that are:

1. **Plumbing:** which means that ROS provides an infrastructure that allows distributed process and programs to communicate between each others.
2. **Tools:** for robot programming, exist an extensive and useful set of tool like simulations, visualization, Graphical user interface etc..
3. **Capabilities** in ROS making it for control, planning, perception, mapping and manipulation. It is possible can make use of existing tools so if you're an expert in planning you don't have to write your mapping addressing yourself to take some of the shelf and develop your own your specific specific area of expertise.
4. **Ecosystem** the ROS software is organized in packages and distribution such as is easily to install and use them. Ros provides also many online tutorials and documentation, thanks to the large community behind it.

[1]ROS Wikipedia: What is ROS?, http://wiki.ros.org/ROS/Introduction

The system was born as a result of various efforts in the robotic field at the Standford University such as the STanford AI Robot (STAIR), Personal Robots that, in 2007, Willow Garage improved significantly these concepts and started creating reliable applications. Besides, several researchers contributed in developed the ROS core and its basic software packages. The whole process was under the permissive BSD open-source license which allows it to gradually become one of the most used platform by the robotic community.

2.3.1 Characteristics

As illustrated brilliantly by Quigley, there are five principles on which ROS has been built and it is thanks to them that ROS has acquired so much popularity among developers.

1. **Peer-to-peer.** Any system that is living in a ROS framework is composed of several processes, potentially on different hosts, that are in communication through a peer-to-peer connection. The Master, that will be addressed, is the one in charge of keeping tracks of the different processes.

2. **Tools-based.** In order to make the system as much modular as possible, it is based on a micro-kernel design that relies on several small tools to build and run the various ROS components. This design loses something in terms of efficiency, but it improves stability and complexity management.

3. **Multi-lingual.** To be sure that each user can write code using its favorite language, ROS is designed to be language-neutral. It currently supports several programming languages as: C++, Python, Octave and others. In order to allow this cross-language environment, ROS uses a simple language-neutral interface definition language (IDL) to describe messages sent between modules. Then, code generators create native implementations for each supported language.

4. **Thin.** Since most of the existing robotics projects contain useful drivers or helpful algorithms, the ROS build system has been planned to perform modular builds inside the source code tree. In this way, the tendency of entangling softwares with their original contexts, making them difficult to reuse, is discouraged. Some clear examples of open-source projects that have been recovered inside ROS are: OpenCV2 for vision algorithms or OpenRAVE 3 for planning algorithms .

5. **Free and Open-Source.** The full ROS code is made publicly available in order to facilitate the debugging procedure. In particular, it is distributed under the BSD license terms, which allows the deployment of both commercial and non-commercial projects.

2.3.2 ROS Concepts

According to the official ROS wikipedia, there are three different levels of concepts: Filesystem level, Computation Graph level and Community level.
Starting with the **Filesystem level**, which embeds mainly disk resources, the following concepts are important:

- **Packages.** Packages are the main unit to organize software in ROS. A package may contain ROS runtime processes (nodes), a ROS-dependent library, datasets, configuration files,anything else that is usefully organized together. Packages are the most atomic build item and release item in ROS. Meaning that the most granular thing you can build and releasea package.

- **Metapackages.** Metapackages are specialized Packages which only serve to represent group of related other packages.

  Package Manifests. Manifests (package.xml) provide metadata about a package, including its name, version, description, license information, dependencies, and other meta information like exported packages.

- **Repositories.** A collection of packages which share a common VCS system. Packages which share a VCS share the same version and can be released together using the catkin release automation tool bloom. Repositories can also contain only one package.

- **Message (msg) types.** Message descriptions, stored in `my_package/msg/MyMessageType.msg`, define the data structures for messages sent in ROS.

- **Service (srv) types.** Service descriptions, stored in`my_package/srv/MyServiceType.srv`, define the request and response data structures for services in ROS.

On the other hand, the **Computation Graph level** consists in the peer-to-peer network of ROS processes that are processing data together. The following concepts, included in the `ros_comm` repository, are relevant:

- **Nodes.** Nodes are processes that perform functions or algorithms designed for a specific robotic components. For instance, there can be a node representing the camera, another representing the microphone and so on. ROS is designed to be modular at a fine-grained scale, this means that a robot control system usually comprises many nodes. They are the core of the graph and they communicate one with another by exchanging messages via Topics and Services. The modularity of the graph, composed of different nodes, makes the whole system more stable since a crashing node is not affecting the whole environment.

- **Master.** The ROS Master acts similarly to a DNS server by providing name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would notable to find each other, exchange messages, or invoke services. It is the only one in charge of controlling the whole system; therefore only one Master at a time can be running in a network, and it can be located in any machine inside it, assuming that the environment is correctly configured.

- **Parameter Server.** The Parameter Server allows data to be stored by key in a central location. It is currently part of the Master.

- **Messages.** Nodes communicate with each other by passing messages. A message is simply a data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as well as arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structs).

- **Topics.** Messages are routed via a transport system with publish/subscribe semantics. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence. The idea is to decouple the production of information from its consumption. Logically, one can think of a topic as a strongly typed message bus. Each bus has a name, and anyone can connect to the bus to send or receive messages as long as they are the right type.

- **Services.** The publish/subscribe model is a very flexible communication paradigm, but its many-to-many, one-way transport is not appropriate for request / reply interactions, which are often required in a distributed system. Request/reply is done via services, which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply. ROS client libraries generally present this interaction to the programmer as if it were a remote procedure call.

- **Bags.**Bags are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms.

Finally, there is the **Community level**, which allows separate communities to collaborate exchanging software and knowledge. The main ROS resources included in that layer are:

- **Distributions.** ROS Distributions are collections of versioned packages that can be installed. Distributions play a similar role to Linux distributions: they make it easier to install a collection of software, and they also maintain consistent versions across a set of software.

- **Repositories.** ROS relies on a federated network of code repositories, where different institutions can develop and release their own robot software components.

- **ROS wiki** The ROS community wiki is the main forum for documenting information about ROS. Anyone can sign up for an account and contribute their own documentation, provide corrections or updates, write tutorials, and more.

2.3.3 ROS visualization (RVIZ)

... in this part will be explained, what is tools and how does it works..

2.4 Virtual Private Network (VPN)

A VPN allows to extend a private network over a public one, allowing users to send and receive data across shared or public networks as if their devices were directly connected to a private one. In fact, inside the VPN system, unique IP addresses are assigned to the devices, as if they were private IP addresses.

When it was first designed, its only purpose was to allow remote access to computers by creating a network tunnel. However, the VPN has been later adopted mostly to mask the original IP address, for security purposes. In few words it works as follows. Usually, when a device asks for Internet connection, it first connects to the Internet Service Provider (ISP), which then forwards it to any website that is required. Therefore, all the Internet traffic passes through the ISP servers and can be tracked by the corresponding ISP. Instead, when using a VPN, a device is connecting to a server run by the adopted VPN provider through an encrypted connection. In this way, the data is secured and, moreover, the IP address cannot be tracked back by the ISP. This stratagem has several consequences:

- The ISP cannot see the data, since they are encrypted, and it cannot know which websites the user is navigating since all the Internet activity is routed through the VPN server. Basically, the ISP knows only that the device is connected to the VPN server.

- The real IP address is hidden behind the VPN server IP address. This means that, unless the VPN server is providing more details, the visited websites cannot track your real IP address back.
- Even if the data are encrypted, the trust is just moved from the ISP to the VPN provider. In fact, while the ISP is totally unaware on what is going on, the VPN provider can have access to all the sensitive data.
- The main drawback of using a VPN is the network latency. It is mainly caused by two factors: data encryption and decryption that require processing power, and the extra distance that must be covered by the data to reach the VPN server.

ROS works just on private network, so if a VPN is used to, all the devices included in the ROS network can be embedded inside it, assigning each of them a unique IP, no matter which is their corresponding LAN. Once that each device is characterized by an unambiguous IP address, it can be used to reference robots inside the ROS environment, simulating a unique network. Adopting the VPN technology, the multi-machine architecture is intrinsically secure as it is tunneled through Internet.

To initiate a VPN architecture, different software solutions exist, in this Thesis the OpenVPN 2.4 software application has been used.

2.4.1 OpenVPN

OpenVPN is an open-source software application used to implement VPN structures for building secure point-to-point or site-to-site connections in routed or bridged configurations and remote access facilities. It is based on a custom security protocol that uses Transport Layer Security (TLS) or Secure Sockets Layer (SSL) cryptographic protocols for key exchange provide different authentication modalities, such as: pre-shared secret key, certificates or username/password system.
For the scope of this Thesis, OpenVPN has been used to build a multiclient-server configuration, in which the server releases authentication certificates for every client, using signature and Certificate Authority (CA).
Several are the reasons for which OpenVPN has been adopted to implement the VPN infrastructure used in this Thesis. It has many features that distinguish it from the other VPN software packages, but the main ones are the following:

- OpenVPN is open-source and, as a consequence, it relies on an efficient developers community. Many times, during the development of the project, support has been asked and it contributed considerably to the success of the application.
- OpenVPN offers an extreme portability, being it compatible with Linux, Solaris, OpenBSD, FreeBSD, NetBSD, Mac OS X and Windows (2000/XP and later versions).
- OpenVPN allows the user to customize significantly the network attributes. Everything from the security countermeasures to the protocol used can be tuned according to the application needs and requirements.

General Concept

Before entering in details with the explanation of how the client/server system has been implemented, some useful concepts have to be addressed in order to allow a better understanding of the work done. In particular, the following ones are of main interest:

- The first clarification is concerning the difference between the Transmission Control Protocol (TCP) and User Datagram protocol (UDP). They are both used to send packets over the Internet and are based on the IP protocol. This means that whether TCP or UDP is adopted, the corresponding packet is sent to an IP address. However, there are some significant differences that play a crucial role when choosing one or the other. The TCP is an end-to-end transmission control protocol, it manages packet ordering, error control, rate control, and flow control based on packet round-trip time. TCP guarantees the arrival of each packet. However, excessive retransmission of TCP in a congested network may introduce undesirable time delays in a networked telerobotic system. On the other hand, UDP behaves differently: it is a broadcast-capable protocol and does not have a retransmission mechanism. Users must take care of error control and rate control themselves. UDP has a lot less overhead compared to TCP. UDP packets are transmitted at the sender's preset rate and the rate is changed based on the congestion of a network. UDP has great potential, but it is often blocked by firewalls because of a lack of a rate control mechanism.

- The second important concepts is the usage of Certificate Authentication (CA) mechanism for improving the connection security. OpenVPN makes usage of the X.509 certificates for client authentication and VPN traffic encryption. Once that the CA is trusted, a Public Key infrastructure (PKI) can be built. When many users are involved, there is much more potential for lost and stolen keys and employee turnover. Therefore, with a properly configured PKI, it is a relatively simple matter to revoke a lost certificate, or that of a departing employee. However, with a single point-to-point link, it often does not make sense to involve the complexity of PKI to protect a tunnel; and pre-shared keys are sufficient to guarantee protection. Finally, after the PKI is setup, it is important to understand the steps that are performed by the client and the server before initiating a trusted connection.

2.5 Operating-System-Level Virtualization

In this part, Operating-System-Level Virtualization features will be introduced...

2.5.1 Containers

...containers are very useful in building applications, in building parallelism and getting things to switch from process to process very fast. They're easy to write relatively, and a great way of packaging ideas into threads that interact... How is possible to turn that idea of containers into a production environment?...

2.5.2 Docker

Docker is a way of making all containers actually sort of uniform following an API effectively. What they do is to wrap up your software in a whole complete container, but, formalized in such a way that has all of the parts that you need to run on all the different systems you're likely to meet. So what it does is to get a complete file system that contains everything to run: your code, your runtime, the system tools, system libraries, anything you would want, and it packages it together in your container along with the code you had written to do all this multithreading.

- **Automates the deployment**
  This guarantees that the software always run the same regardless of which environment it's running on. So now your dockers actually sort of operating in multiple different ways, it's not only allowing you to write that thread software put it in the container. But it's building

that container so it will interrupt right and also to different architectures, and all sorts of different hardware. It not only does that, but it also makes life easier because now when is needed to put a software on a machine, instead of having to write scripts and code to build the system there, it automates the deployment, it will actually take all those application inside the container and put them inside the machine, links them to the right libraries for that machine.

- **Additional layer of abstraction**
  It abstracts the operating system and the libraries, and makes them all look the same. And the way it does it is to introduce it's own set of APIs that you write to as you write your Docker code. But it then interfaces those APIs down to all the different operating systems that you like to run your Docker system on. So, for example, it affectively does virtualization on top of Linux. Doesn't matter what the Linux is, what sort of drives it is,who actually is Red Hat or Ubuntu. What it's going to do is allow your threads to run on those systems without having to do anything, and it automate it.

.

A bit of history. Its origins are way back in the days of had open things like that, we were building platforms as a service. People wanted to build all sorts of unique projects. Solomon Hykes started of in Docker. He was looking at building something called dotCloud, a platform as a service company. The first real effective version of this was released in 13th March 2013. It provide a libcontainer library. It was written in the Go programming language. Improving that notion of providing an environment field threads, that's portable and automated. So the team putting it together, docker team and had a wide spread industrial support. And so the docker container system that you've got actually will map to a huge variety of different Linux systems, and it'll be very reliable, very flexible.

Docker Structure

Docker has as said before, a packaging aspect. The way you build those Docker containers is exactly the way you saw for building alternate containers, but also contains libraries.

- libcontainer is one of them...
- Libvirt, which virtualizes the Linux kernel, so you don't have to know the details about the kernel. And it also separates you from other Dockers running on top of the kernel. It operates a bit like virtual machines, what you care about is being able to organize it the right way, and that is what Docker allows you to do...
- There is LXG which is the container mechanism...
- There is a system daemon spawn which actually manages how docker is set up and what runs...

So those are the components and they all sit on top of your whichever Linux kernel and make everything work. And it doesn't really matter if you want to use SE Linux because you want security from the outside world you can use it. If you want to use capabilities, if you want to use cgroups, or if you want to use Netlink. It all maps from your threads through the docker to the Linux kernel.

As an example, following the right scheme 2.3, the applications (App 1,App 2,App 3) would sit on top of the normal set of libraries. Going on App 1, it's going to need certain libraries to execute

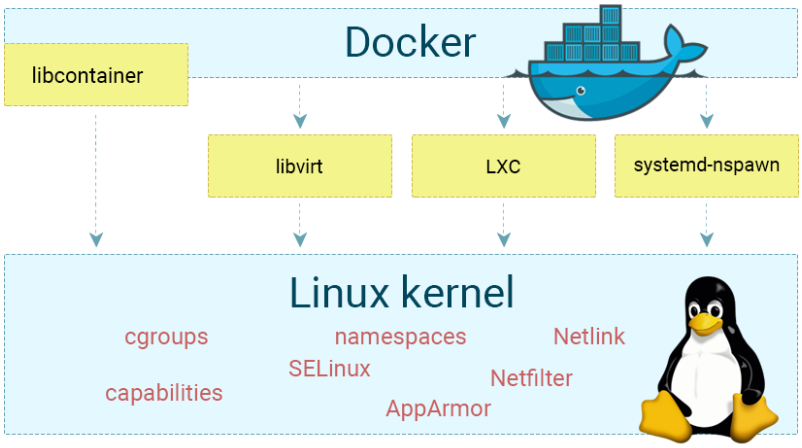

Figure 2.2: Docker Internal structure .

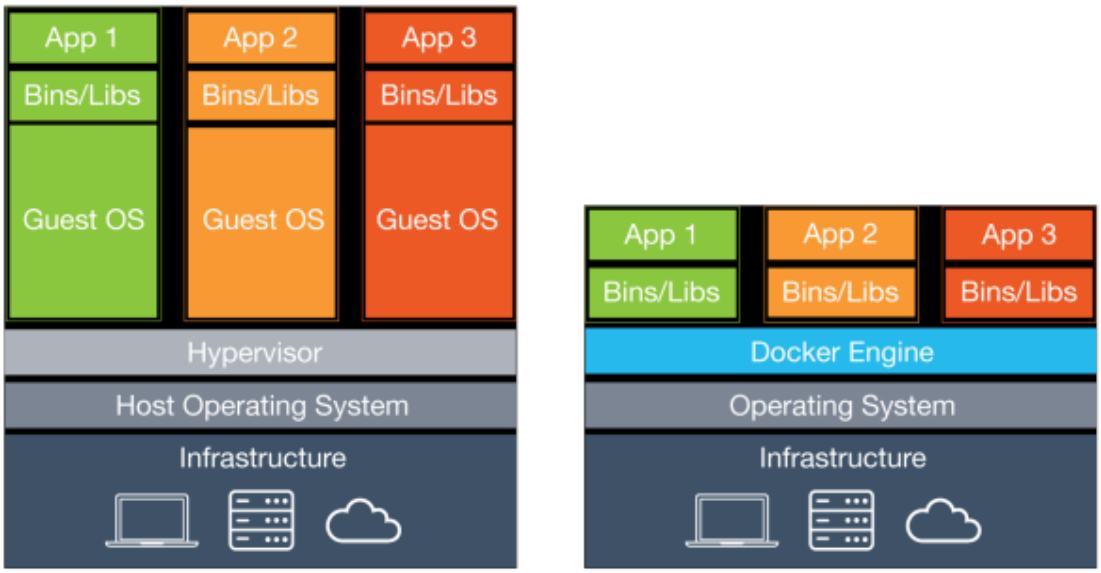

Figure 2.3: Docker vs VM

the system code, and it's going to need certain libraries to map it down on to the system level. And they're might be binaries (Bins) that are needed to run like file systems and so on, that are specific to your particular set of trade, and so that will be contained in the bins. So, if the package App1 + Bins/Libs is what we are interested in having run and it's all sort of virtualized for Linux, then the Docker engine takes that code, maps it to whatever operating system is installed down and the infrastructure under that the operating system really supports. Morevore, while App 1 is running, is possible to start and run a different application, or the same application, several times, that works fully indipendently from each others.

So this overall is real sort of interesting lightweight mechanism to do things. It's lightweight both threads and from the point of view of development.

Docker Architecture

What we described was we take out threads and so on and we turn them into container.Then what's the story, how does it get automated and so on? This particular picture delves into how Docker operates.

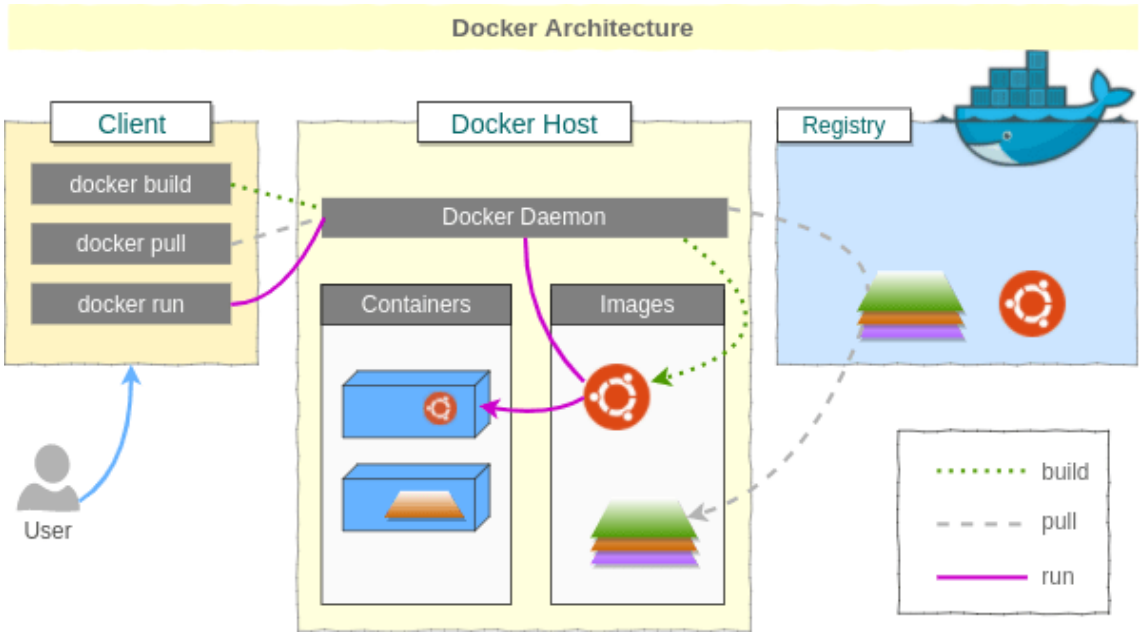

Figure 2.4: Docker Architecture.

What happens is that the user integrates everything together into a DockerFile that contains the file system, the libraries, the interfaces and so on and there is a bunch of source code repositories that are used to help build that. So, user got a system with the relative threads, that will be on your sort of build system, your development environment. You build it and create, through the Docker Daemon, a effectively container, containing all those parts.

So now is possible to transfer that to other machines. Now typically, what you would do is to hook that and yet a different container, a registry and why is it a registry? So that other users can actually look up that particular component and pulled it down to whichever of the cluster nodes that they want it to run on or maybe all of them. So this image registry is sort of operating like a server of Dockers, of whatever people have created and installed in that particular system. User can do that under program controller or can issue the commands to do that. When the user pull a container down and run it, it will be executed on top of his associate Linux images, almost like another VM. But of course it's all running at user lever, so working but every user has his threads all operating asynchronously. They can use all of the packages done at Linux, so they get some concurrency and some parallelism and this is all self contained...

Chapter 3

PARLOMA

PARLOMA is a project developed by L. Russo and colleagues. As the PARLOMA website is stating: "The PARLOMA project is aimed at creating a robotic system to allow remote communication between two deafblind people, a deafblind person to a deaf person and a deafblind person to a hearing person with a knowledge of sign language". In order to guarantee such a communication, the project relies on an anthropomorphic robotic hands, which replays the Sign Language gestures. Since the system is based on ROS and requires a remote multi-machine architecture, the networks developed in this Thesis can be used to build it.

3.1 PARLOMA: State of the art

The PARLOMA project was born from the collaboration between Politecnico di Milano and Politecnico di Torino, thanks to the Alta Scuola Politecnica (ASP). It was started by a team of students from the VIII ASP cycle, about six years ago, and it is still in development under the sponsorship of the Ministry of Education, University and Research (MIUR).
The project wants to give deafblind people the possibility to communicate remotely, which at the moment is not provided by any device. Deafblind people communicate through the Sign Language (SL) as long as they are able to see, on the contrary, they use the Tactile Sign Language (TLS). Even if this way of communicating allows them to entertain an effective conversation, it has huge limitations.

Especially, the following are the main problems:

- both the interlocutors (signer and receiver) must be in the same place when communicating, since they need to keep their hands in touch;
- only one-to-one communication is possible, since the receiver must maintain the contact with the hand of the signer.

In order to overcome these strong limitations, PARLOMA tries to allow both remote communication, thanks to the usage of a cloud-based robotic platform, and multiple conversation, by providing multiple devices that are remotely replying the signer gestures that can be used by several receivers. Figure 3.1 shows how PARLOMA works.

Although at first sight the project could seem very different with respect to a telepresence application, it makes usage of the very same technologies.

The three main technologies on which PARLOMA is based are:

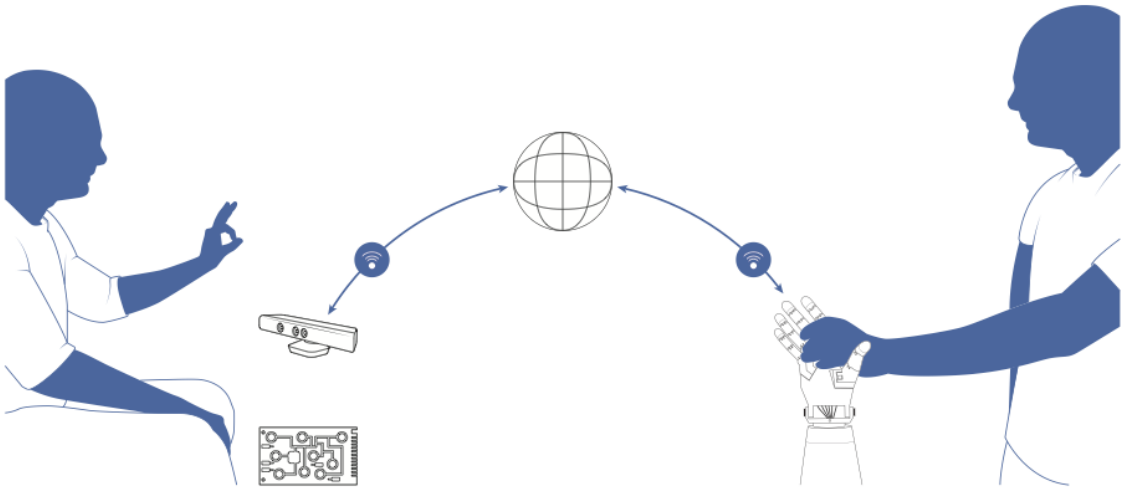

Figure 3.1: Illustration of how PARLOMA operates

1. **Robotic Operating System (ROS)**, used as framework for managing the communication between devices as well as the hardware complexity.
2. **Amazon Web Service (AWS)**, used to host the cloud computing environment.
3. **Raspberry Pi**, used as processing unit of the system, connected with the robotic hand. It is in charge of receiving the commands from the cloud platform and controlling accordingly the robotic hand.

The operations performed by the PARLOMA device can be classified in three main groups:

- **Hand tracking**, performed by means of the Leap Motion device.
- **Data transmission**, from the signer to the receiver, that must be guaranteed to be private and secure.
- **Gestures reproduction**, by means of a robotic hand that moves accordingly to the signs performed by the signer.

...

3.2 PARLOMA: Open Issues

Among these three areas, the work done by the Author interests the data transmission phase, where a secure channel is needed to forward the information from the signer device to the cloud platform based on ROS. For this purpose, in the next section the architecture conceived to ensure the successful communication between devices will be addressed, and the Author contribution will be discussed.

For the sake of simplicity, the architecture will be split in two parts: the Communication Architecture, which illustrates the overall communication between the input device and the output one, and the Cloud Interface Architecture, which describes in details how the actions necessary to guarantee the communication are implemented.

Communication Architecture. This architecture is responsible of ensuring the data streaming from the input device to the output one, by means of the cloud platform. Therefore, it assumes that the three machines involved (input device, output device and remote server) are already connected in a common network and are able to reach each other.

Three main modules compose the architecture:

1. Input module, on the left, acquires in real-time the signer device input measurements and processes them in order to perform the hand tracking. It embeds the Input Driver, which is the actual sensor driver, and the ROS Driver, which converts the measurements in ROS messages, in order to send them to the following unit.

2. Remote Computation module, in the center, which guarantees the communication between the two external modules through a VPN channel and completes the hand tracking procedure.

3. Output module, on the right, receives the hand tracking information from the remote computation module via standard ROS messages and, complementary to the input module, moves the receiver device by means of the ROS Driver and the Output Driver.

.....

Chapter 4

Contribution to PARLOMA

In this Chapter, the main part of the Thesis project will be addressed: the extension of the PARLOMA features, allowing to use different machines as input/output modulus that are connected to different ROS networks through Internet. This concept is a key-point of the Cloud Robotics paradigm and it is a crucial requirement to exploit all the cloud potentialities.

Before this Thesis work, the PARLOMA project gave the possibility to initiate only local ROS networks. This means that all the I/O modulus taking part of the network had to be connected to the same LAN to be identified by the ROS *Master*. Thanks to the Author contribution, this restriction has been overcome, allowing the expansion of the same ROS architecture over I/O modulus connected to different LANs.

This improvement greatly enhances the potentialities of the project, giving users the possibility of making usage of several new features. In particular, there are a lot of useful advantages, reusable also on others kind of ROS-based systems.
The main advantages are the following:

- First, **telerobotics** can be seriously implemented. The term indicates the combination of *teleoperation* and *telepresence* concepts in a robotic system that can be remotely controlled.
- Second, **distributed computing** can be fully adopted. Once that there are no boundaries related to the local area, the user is free to use any powerful machine to run the heavy computation, without minding where it is located. It is sufficient to have a stable Internet connection to log the device into the extended ROS network, and to use it in order to off load huge computations from the robot-hand. For example, considering huge computer vision algorithms, they could be run on a remote sever connected to the robot by means of the platform, moving all the computational efforts away from the local robot hardware.
- Third, **collective robot learning** is promoted. Once that the architecture is initiated, several robots from all over the world can connect to it, sharing information and feedbacks. In this way, the cloud database takes advantage not only of the information coming from one robot, but from a set of robots acquiring useful data. For example, think about a situation where a robot is depending on the action of another one, i.e., they need to exchange information and tasks, but they are in different environments.

To implement such an extension of the local network, a multi-machine architecture had to be set up. The Robotic Operating System (ROS), which has been used for this project, provides several tools to start a system using multiple machines. However, as described in details in the official

ROS guide[1], in order to guarantee a correct initiation of the network, all the included machines must be able to satisfy these two requirements:

1. First, there must be complete, bi-directional connectivity between all pair of machines, on all ports.
2. Second, each machine must advertise itself by a unique name that all other machines can resolve.

In order to satisfy these requirements and to create the VPN-based ROS network different tools/software have been used, and different source code has been written ....

1. DOCKER
   - Server Dockerfile: ....
   - Client Dockerfile: ....
2. OpenVPN
   - Client/Server configuration files: ....
   - Client/Server certificates: ....
3. ROS
   - ROS Master ...
   - ROS publisher "talker" node: ....
   - ROS subscriber "listener" node: ....

4.1 DockerFile: Build & Run

.....
As explained in the official Docker documentation[2] the following commands allow to make a Dockerfile

- **FROM**
  The FROM instruction initializes a new build stage and sets the Base Image for subsequent instructions. As such, a valid Dockerfile must start with a FROM instruction. The image can be any valid image – it is especially easy to start by pulling an image from the Public Repositories. FROM can appear multiple times within a single Dockerfile to create multiple images or use one build stage as a dependency for another. Simply make a note of the last image ID output by the commit before each new FROM instruction. Each FROM instruction clears any state created by previous instructions.

  Optionally a name can be given to a new build stage by adding AS name to the FROM instruction. The name can be used in subsequent FROM and COPY –from=<name|index> instructions to refer to the image built in this stage.

  The tag or digest values are optional. If you omit either of them, the builder assumes a latest tag by default. The builder returns an error if it cannot find the tag value.

[1]ROS multi-machine network setup, http://wiki.ros.org/ROS/NetworkSetup

[2]Dockerfile reference, https://docs.docker.com/engine/reference/builder/

- **RUN**
  The RUN instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile. Layering RUN instructions and generating commits conforms to the core concepts of Docker where commits are cheap and containers can be created from any point in an image's history, much like source control.

  base image that does not contain the specified shell executable. The default shell for the shell form can be changed using the SHELL command. In the shell form you can use a (backslash) to continue a single RUN instruction onto the next line. For example, consider these two lines:

  Note: Unlike the shell form, the exec form does not invoke a command shell. This means that normal shell processing does not happen. For example, RUN ["echo", "$ HOME"] will not do variable substitution on $HOME. If you want shell processing then either use the shell form or execute a shell directly, for example: RUN ["sh", "-c", "echo $ HOME"]. When using the exec form and executing a shell directly, as in the case for the shell form, it is the shell that is doing the environment variable expansion, not docker.

  The cache for RUN instructions isn't invalidated automatically during the next build. The cache for an instruction like RUN apt-get dist-upgrade -y will be reused during the next build. The cache for RUN instructions can be invalidated by using the –no-cache flag, for example docker build –no-cache.

  See the Dockerfile Best Practices guide for more information.

  The cache for RUN instructions can be invalidated by ADD instructions. See below for details.

- **CMD**
  The main purpose of a CMD is to provide defaults for an executing container. There can only be one CMD instruction in a Dockerfile. If you list more than one CMD then only the last CMD will take effect. These defaults can include an executable, or they can omit the executable, in which case you must specify an ENTRYPOINT instruction as well. When used in the shell or exec formats, the CMD instruction sets the command to be executed when running the image. If the user specifies arguments to docker run then they will override the default specified in CMD Don't confuse RUN with CMD. RUN actually runs a command and commits the result; CMD does not execute anything at build time, but specifies the intended command for the image.

....
now that we have the basics knowledge on what a Dockerfile is composed, let's see how to use these commands...
....

4.1.1 Server Dockerfile

```
1 FROM osrf/ros:indigo-desktop-full
2 RUN
3 svn checkout https://github.com/creos92/thesis.git/trunk/Server
4 svn checkout https://github.com/creos92/thesis.git/trunk/inmoov_ros
5 apt-get update && apt-get install build-essential checkinstall libssl-
    dev liblzo2-dev libpam0g-dev wget -y
```

```
6 wget https://swupdate.openvpn.org/community/releases/openvpn-2.4.4.tar.
    gz && sudo tar zxvf openvpn-2.4.4.tar.gz && cd openvpn-2.4.4 && ./
    configure && make && checkinstall -y
7 /bin/bash -c "source /opt/ros/indigo/setup.bash && cd /inmoov_ros &&
    catkin_make"
8 CMD cd inmoov_ros && chmod +rx start_s.sh && sync && ./start_s.sh
```

Listing 4.1: Server Dockerfile

The command :

```
1 FROM osrf/ros:indigo-desktop-full
```

is used to sets a ROS base Image. In this way, when we will start the docker containers ROS will be already installed.

The commands:

```
3 svn checkout https://github.com/creos92/thesis.git/trunk/Server
4 svn checkout https://github.com/creos92/thesis.git/trunk/inmoov_ros
```

are used to download on top of the ROS layer,the OpenVPN server side configurations and license files explained in the section ..., and the ROS Package that allow the hand virtualization trough RVIZ.

The commands:

```
5 RUN /bin/bash -c "apt-get update && apt-get install build-essential checkinstall lib
wget -y"
5 RUN /bin/bash -c "wget https://swupdate.openvpn.org/community/releases/openvpn-2.4.
```

are used to download and install the OpenVPN software, useful to start later the VPN between servers and clients

The command :

```
6 RUN /bin/bash -c "source /opt/ros/indigo/setup.bash && cd /inmoov_ros && catkin_mak
```

is used instead to source the setup.bash file and to install the ROS_Packake before downloaded.

The last command:

```
6 CMD cd inmoov_ros && chmod +rx start_s.sh && ./start_s.sh
```

represent the first executed command when the docker container will be started, it make the start_s.sh executable and execute it.

Here, the start_s.sh file is listed.

```
1 source ./devel/setup.bash
2 chmod +rx ./start_rviz.sh ./start_talker.sh /Server/certificati/server/start_server.sh
3 cd /Server/certificati/server && ./start_server.sh &
4 export ROS_IP=10.8.0.1
5 export ROS_MASTER_URI=http://10.8.0.1:11311
6 cd /inmoov_ros && ./start_rviz.sh & ./start_talker.sh
```

Listing 4.2: start_s.sh file

- The first command source the setup.bash file that means merely to add environment variables to the path allowing the new ROS Package to work correctly.
- The second command make executable the .sh files.
- The third command execute the start_server.sh that basically start the server service with this command

  ```
  openvpn --config server.conf
  ```

  the server.conf files is already explained.
- the fourth and fifth commands are used to interface the ROS server with ROS client, exporting his own ip address and identifying which is the Master node IP address. In this case they corrispond because the ROS MASTER is executed on the Server side.
- The last command execute the following script files
  1. start_rviz.sh that basically start the Rviz that is the 3D visualization tool for ROS, with the hand model showing in real time the hand movement.
  2. start_talker.sh that starts the python script:
     - subscribing in `/output/serial_topic`, that is the topic that contains the information about how tha hands has to move
     - publishing on joint_command topic

4.1.2 Client Dockerfile

```
1 FROM osrf/ros:indigo-desktop-full
2 RUN       \
3         svn checkout https://github.com/creos92/thesis.git/trunk/Client &&\
4         svn checkout https://github.com/creos92/thesis.git/trunk/inmoov_ros && \
5         apt-get update && apt-get install build-essential checkinstall libssl-dev libl
  wget -y && \
6         wget https://swupdate.openvpn.org/community/releases/openvpn-2.4.4.tar.gz && s
7         /bin/bash -c "source /opt/ros/indigo/setup.bash && cd /inmoov_ros && catkin_ma
8
9 CMD cd inmoov_ros && chmod +rx start_c.sh && sync && ./start_c.sh
```

Listing 4.3: Client Dockerfile

4.1.3 Docker start configuration

4.2 OpenVPN: Certificate and configuration files

Usually, the IP addresses of the machines are used as unique names, which means that all the robots must be connected to the same LAN in order to identify each other. Therefore, if the ROS network has to be extended over different LANs, a workaround is needed to configure a network that can be seen by ROS as a local one, even if actually it is not. In this way, the capabilities of ROS can still be used, together with the potentialities of a distributed network.

There are several alternatives to overcome ROS limitations, in particular an option is based on the Virtual Private Network (VPN) concept. A VPN as explained in the section 2.4.1, allows to

extend a private network over a public one, allowing users to send and receive data across shared or public networks as if their devices were directly connected to a private one. In fact, inside the VPN system, unique IP addresses are assigned to the devices, as if they were private IP addresses. In this way, all the devices included in the ROS network can be embedded inside the same VPN, assigning each of them a unique IP, no matter which is their corresponding LAN. Once that each device is characterized by an unambiguous IP address, it can be used to reference robots inside the ROS environment, simulating a unique network.

From the ROS point of view, this architecture is simpler and it is characterized by one single *Master* that is running in one of the machines inside the network.It provides a quite good level of flexibility. Moreover, adopting the VPN technology, the multi-machine architecture is intrinsically secure as it is tunneled through Internet. To initiate the VPN architecture, the OpenVPN[3] software application has been used. The detailed configuration of the VPN system is described in Section 4.2.

Then, once that the VPN was correctly built, the next step was to set up a ROS architecture running inside the VPN system. With this aim, the ROS *Master* has been located remotely, in order to provide the highest degree of versatility and flexibility. Further details on how ROS has been configured to run inside the VPN architecture are provided in Section 4.3.

Finally, after the completion of the environment setup, several tests have been run in order to evaluate the network latency of the VPN ROS network with respect to a Local ROS one and a ROS network running on a single machine. These tests are addressed in details in Section **??**, where also some comments are provided.
OpenVPN is a great tool, which allows several different configurations. Each different model can be then customized in terms of security, transmission protocol and others. Since the possibilities are several and the author had no experience in using this software, the book *Mastering OpenVPN* [1] has been widely consulted and most of the information come from it.

The first thing to do when setting up a VPN is to choose the most suitable deployment model. In this case, since the configuration required must be implemented between one server and several clients, the model *client/server with tun devices* has been depicted. After the model choice, several steps are required to build up a functioning environment; however they will be just listed below and only the server configuration file and the client configuration file adopted will be addressed in details. The different procedures are the following:

- Set up the CA directory and configure the CA variables;
- Build the Certificate Authority;
- Create the Server Certificate, the Server Key and the Encryption Files;
- Generate a Client Certificate and a Key Pair;
- Configure the OpenVPN service by copying all the previous files in the appropriate directories;
- Create the Server configuration file;
- Create the Client configuration file (one for all clients or one for each client);
- Start the service.

[3]OpenVPN,https://openvpn.net/

4.2.1 Certificate files

4.2.2 Server configuration file

. The server configuration file is showed in Listing 4.4. It basically defines all the options that must be taken into account when starting the service, together with some particular customizations. First of all, note that the option *proto tcp* and *dev tun* are used instead of their counterparts, i.e. *proto tcp* and *dev tap*, because they cause less overhead in the VPN tunnel. In particular, as explained in Subsection **??**, the TCP protocol is more robust, also if a bit less efficient. That is why, for this kind of application it is generally preferred, as network latency is considered a priority with respect to possible packages lost. Another important settings are those in lines *6-9*, which are indicating the authentication files needed to establish a secure TLS connection. Finally, another useful information is the one of line *13* that indicates which IP addresses will be assigned by the VPN. In this case they will be of the form *10.8.0.x*. The other parameters are less important and will not be addressed in details as they are not affecting crucially the functioning of the network.

```
1  port 1194
2  proto tcp
3  dev tun
4  sndbuf 0
5  rcvbuf 0
6  ca ca.crt
7  cert server.crt
8  key server.key
9  dh dh.pem
10 auth SHA512
11 tls-auth ta.key 0
12 topology subnet
13 server 10.8.0.0 255.255.255.0
14 keepalive 10 120
15 cipher AES-256-CBC
16 comp-lzo
17 user nobody
18 group nogroup
19 persist-key
20 persist-tun
21 status openvpn-status.log
22 verb 3
23 crl-verify crl.pem
```

Listing 4.4: Server configuration file.

4.2.3 Client configuration file

For this Thesis application, two client configuration files have been produced: one for the desktop PC and another one for the Raspberry. Both of them are tailored to the same server, whose configuration file has been explained before. Therefore, since they are really similar, only the one hosted by the Raspberry will be showed in details in Listing 4.5.

The first line is the most important since it states that the device will be used as a client in a *client/server* configuration. Then, as done before, the protocol and the device typology are

defined. Another crucial line is the number *6*, which is indicating the remote address of the server and it is essential for the connection to be established.

Finally, another main difference that characterizes this file with respect to the server one is that the authentication files are not indicated through the corresponding paths, but they are included in the configuration file itself. This choice has been adopted because, in view of future developments, if more clients have to be initiated by the server side, then just one file needs to be transferred, i.e., the client configuration file, which includes all the security information. On the contrary, if the information was not embedded in the configuration file, the separate files (*ca.cert*, *client.cert*, *client.key* and so on) must be sent together with the configuration file, increasing the probability of making errors. Note that a file including also the security information has the extension *.ovpn* instead of the usual *.conf*.

```
1  client
2  dev tun
3  proto tcp
4  sndbuf 0
5  rcvbuf 0
6  remote 130.192.163.200 1194
7  resolv-retry infinite
8  nobind
9  persist-key
10 persist-tun
11 remote-cert-tls server
12 auth SHA512
13 cipher AES-256-CBC
14 comp-lzo
15 setenv opt block-outside-dns
16 key-direction 1
17 verb 3
18 <ca>
19 -----BEGIN CERTIFICATE-----
20 .........................
21 -----END CERTIFICATE-----
22 </ca>
23 <cert>
24 Certificate:
25     .........................................
26     Signature Algorithm: sha256WithRSAEncryption
27         .....................................
28 -----BEGIN CERTIFICATE-----
29 .........................
30 -----END CERTIFICATE-----
31 </cert>
32 <key>
33 -----BEGIN PRIVATE KEY-----
34 .........................
35 -----END PRIVATE KEY-----
36 </key>
37 <tls-auth>
38 -----BEGIN OpenVPN Static key V1-----
```

```
39 ..........................................
40 ------END OpenVPN Static key V1------
41 </tls-auth>
```

Listing 4.5: Client configuration file. Note that all the sensible information have been substituted by dotted lines, whereas in the original file obviously they are not.

Service initiation. After the two configuration files are correctly hosted in the corresponding devices, the service can be initiated and the VPN tunnel created. In order to establish the connection, only one bash command is needed: *sudo openvpn client.ovpn*. Then, if no errors are present, the connection is built. Note that, during the development of this phase, most of the problems came from the configuration of the part that was acting as a VPN server. In fact, it had some default firewall settings that were blocking the connection with the VPN tunnel.

Automated script. Once that the service was correctly functioning, an automated method has been developed in order to speed up the procedure and make it simpler. After some research, an enlightening bash script[4] has been found on GitHub that automatically produces the required files. It works as follow. The script must be hosted by the server and the first time that is launched it produces the server configuration file, together with the CA file. Then, every time that it is run it produces a client *.ovpn* file that can be customized for what concerns the protocol to be used and other less important parameters, saving all the time needed to build manually the file. It is worth of mention since, in case of future developments, it will be used to automate the production of client configuration files.

4.3 ROS: Multi-machine Architecture

After having set up the VPN system, the integration with the Robotic Operating System (ROS) has been performed. As anticipated in the introduction of this Chapter, the VPN purpose is to allow the usage of ROS across different networks. In particular, the IP addresses assigned by the VPN can be used to identify the different machines inside the architecture, forming a single independent system. Before going in details about how the architecture has been implemented and how ROS has been combined, a general overview of multi-machine ROS networks is presented.

As mentioned before, to build a ROS multi-machine network two conditions must be fulfilled:

1. First, there must be complete, bi-directional connectivity between all pair of machines, on all ports.
2. Second, each machine must advertise itself by a unique name that all other machines can resolve.

For the fulfillment of the two requirements, the VPN plays a crucial role. In fact, once that OpenVPN is running, each machine is able to connect with the others through the VPN IP address. Hence it can be used to identify univocally the machines inside the ROS network.

- The first condition can be easily verified from a Linux terminal, by using the command *ping* followed by the VPN IP adress of the other machine, where the server is assigned the address

[4] GitHub open-vpn install bash script, https://github.com/Nyr/openvpn-install

10.8.0.1 and a generic client machine the address *10.8.0.2*. As can be seen from the figure, they are able to ping each other successfully.

- Then, to satisfy the second requirement, again the IP provided by the VPN is used. In fact, when a ROS node advertises a topic, it provides a hostname:port combination (a URI) that other nodes will contact when they want to subscribe to that topic. It is important that the hostname that a node provides can be used by all other nodes taking part in the network to contact it. The name can be set explicitly by means of the *environment variables* provided by ROS; in particular one between the *ROS_IP* and the *ROS_HOSTNAME* must be defined. Therefore, for the aim of the proposed application, the *ROS_IP* of each machine in the network is set with the corresponding VPN IP address. Recalling the example where the generic client has the address *10.8.0.2*, it will be sufficient to run the ROS command *export ROS_IP=10.8.0.2* to set the right name within the network.

4.3.1 Server side

Entering more in details on the application developed, the ROS network adopted will be presented and the main aspects discussed. Since ROS is based on the idea of a *Master* that is working as a server, keeping track of the several nodes and topics inside the network, the same concept has been applied to the project. In order to build a multi-machine infrastructure as flexible as possible, the *Master* has been placed in a remote machine; in particular, in the same device that works as a server inside the VPN architecture. In this way, the VPN server and the ROS *Master* become one entity that is both providing VPN connection to clients, together with ROS potentialities.

To host them,whatever architecture can be used, thanks to Docker. Once the access has been completed, the *Master* can be initiated and its IP, indicated as *ROS_MASTER_URI*, can be exported, together with the *ROS_IP* of the machine, in order to set up the multi-machine framework.

In the architecture used in this Thesis, there were three devices involved: a "Docker server", and two "Docker client".

In the following list, the commands needed for their initialization are shown in details.

- The **Docker Server** is working as a *Master* with IP address *10.8.0.1*. The ROS commands needed to configure it are:

  1. *source ~/catkin_ws/devel/setup.bash*
  2. *export ROS_IP=10.8.0.1*
  3. *export ROS_MASTER_URI=http://10.8.0.1:11311*
  4. *roscore*

4.3.2 Client side

After the *Master* initialization, the other machines can be configured to enter in the same ROS network. To this purpose, assuming that the VPN connection is established, they have to export their *ROS_IP* as well, followed by the *ROS_MASTER_URI* that is the same for all the client machines and it refers to the one of the *Master*.

- The **Docker client on Desktop PC** is working as a VPN client with IP address *10.8.0.2*. The ROS commands needed to configure it are:

1. *source ~/catkin_ws/devel/setup.bash*
2. *export ROS_IP=10.8.0.2*
3. *export ROS_MASTER_URI=http://10.8.0.1:11311*

- The **Docker client on Raspberry Pi 3 B** working as a VPN client as well with IP address *10.8.0.3*. The ROS commands needed to configure it are:

1. *source /opt/ros/indigo/setup.bash*
2. *export ROS_IP=10.8.0.3*
3. *export ROS_MASTER_URI=http://10.8.0.1:11311*

4.4 Other configuration files

....In this section will be explained the script files that automate the "call initiation"...

Chapter 5

Experiments

..Two actual case studies making usage of the VPN Multi-Machine ROS network developed are addressed. Two configurations will be tested: single client/server and multiple client/server....

5.1 Experiment 1:

....

5.2 Experiment 2:

....

5.3 Remarks

....

Chapter 6

Conclusions

6.1 ROS Multi-Machine

6.1.1 Results

6.1.2 Future Developments

6.2 Final Remarks

From this collaboration, the author had the chance to learn how to use several new technologies that were involved in this Thesis project. In particular, the following have been used for the first time:

- **Linux** operative system has been widely used to program the Raspberry and to work with Docker.
- **Python** programming language has been used both to implement the WiFi configuration and to develop the robotic application.
- **Raspberry** computer has been used as hardware environment during the Thesis development.
- **Docker** software has been used to set up server and client on whatever architecture.
- **ROS** has been used for the extended robotics network built and for the telepresence application.
- **OpenVPN** software has been used to set up the VPN architecture for the ROS extended network.

Bibliography

[1] E. F. Crist and J. J. Keijser. *Mastering OpenVPN*. PACKT Publishing, 2015.

List of Figures