Study of Robotics Simulator

Submitted By

Karan Patel (Student ID - 202112074)

Honey Patel (Student ID – 202112096)

Submitted To:

Department of Robotics and Intelligent Systems.

AT DAIICT 2023

Summary

Simulation software is becoming an increasingly important tool for automation systems both in industry and research. Within the field of mobile robotics, it is used to evaluate the performance of robots regarding localization, motion planning and control. In order to evaluate the performance of a mobile robot based on a simulation, the simulation must be of sufficient precision and accuracy. A commonly used middleware in robotics with increasing usage and importance is ROS (Robot Operating System). The community and the number of robots in industry and research that are using ROS is growing. The performance of Unity as a simulator in the ROS environment should be tested in this thesis work. While testing the performance of the simulator, its ability to adapt to different scenarios and applications should be rated.

Overall, Unity was rated to be suitable for the simulation of mobile robots in robotics research if no accurate simulation of the robots' properties is needed. Of major importance when conducting simulations with mobile robots in Unity are the used robot model and the chosen parameters in the simulation. Only if the robot model is validated, the results can be judged accordingly. To reduce the effect of the parameter settings in Unity on the results, no setups to solely examine the robot's hardware should be chosen. Instead, the simulations can be used for testing, improving and evaluating control algorithms on the robot. A huge opportunity is seen in the usage of automatically and synthetically generated simulation data for teaching deep learning algorithms.

Preface

This summer internship project is done under Prof. Tapas Kumar Maiti with the co-operation of Department of Robotics and Intelligent Systems. We would like to thank Prof Tapas Kumar Maiti helping us with the measurements and sharing their knowledge with us.

Introduction

In recent years, mobile robots are becoming more and more common in industry and private households. According to the definition of the International Organization for Standardization, a robot is an actuated mechanism programmable in two or more axes with a degree of autonomy, moving within its environment, to perform intended tasks. A mobile robot is defined as such a robot which is able to travel under its own control. In industry, self-driving vehicles are used in factories to transport material or machines between processes efficiently. In the sector of service robotics in private households, mobile robots are most commonly used as vacuum cleaners, lawnmowers or floor and window cleaners. They assist the customers to clean their house and garden efficiently. When developing those mobile robots or planning their usage, simulation can be used in various stages. Simulation can be a huge advantage when real robot prototypes or products are not available or cannot be used due to other circumstances. During the development, simulation can be used to assess the basic hardware functionality. In addition to that, the algorithms for localization, motion planning or control can be tested, improved and integrated continuously. Simulation software can be of great advantage not only if no real hardware is available, but also if the scenario is difficult to test in reality or if the quantity of required experiments/iterations is too large to be efficiently tested in reality. In those cases, the simulation can be used to judge the performance of the robot and/or its concept. This usage can increase the efficiency and decrease the costs of the development. Apart from all the advantages, the simulation will always stay a simplified depiction of the reality. Assumptions have to be made when simulating real environments and thus decreasing the accuracy of the simulation. The simulation can only be used in the previously mentioned setups if the simulation is accurate enough to let the user draw conclusions out of the results. If the measuring uncertainty is greater than the accuracy that is needed for the evaluation, using the simulation is not applicable. Unity is a popular game-engine and widely used for the development of video games. It comes with a high-functional and user-friendly graphical user interface. It uses PhysX by NVIDIA as physics engine to simulate the behavior of objects in regard to their physical environment and forces acting upon them. Unity’s user-friendly interface, its flexibility and its functions in the area of artificial intelligence and human machine interaction are beneficial for the choice of a mobile robot simulator. It can be extended with toolkits that enable the development of learning environments with a Python API in Unity. Complex sensors and physical environments can be modelled and dynamic multi-agent interaction is supported.

Regardless of the simulators in robotics, the accuracy and performance of these simulators in regard to reality still has to be evaluated. It is of essential importance to judge the limitations and performance of those simulators in order to use simulation results. Only if the deviation to reality is known, the results can be used within reason. To assess the limitations and performance of simulators is a challenging and ongoing task. It is highly dependent on the settings of the simulators as well as the used physics engines and their implementation into the simulators. The simulators and some of the physics engines are still under ongoing, intensive development, which makes the evaluation even more challenging.

ROS

ROS (Robot Operating System) is – despite its name – not an ordinary operating system but a middleware working on top of a host operating system and providing services similar to those of an operating system. It is designed to be free, open-source, multilingual, thin, peer-to-peer and tools-based. It has been developed during projects at Stanford University and Willow Garage and is nowadays maintained by Open Robotics. One of the main goals of ROS is to reuse and share code in robotics research and development. In that way researchers could collaborate with each other globally on the same framework by using and extending code of other researchers to solve common problems. The ROS framework can be used with Python, C++ and Lisp by default. With additional implementations, any modern programming language can be used, which makes it suitable for diverse usage. For programming in ROS with Python, C++ and Lisp the libraries rospy, roscpp and roslisp can be used, respectively.

The documentation of ROS divides ROS in three different levels of concept: the ROS filesystem level, the ROS computation graph level and the ROS community level. On the filesystem level, ROS is built up of types like packages, repositories and message types. The main unit for organizing software in ROS are packages. They can include nodes, libraries, configuration files or datasets which are dependent on each other and should therefore be organized together. One or more packages can be collected and grouped into repositories. Message types describe the structure of messages that can be published to and subscribed from topics in ROS. There exist primitive message types, e.g., bool, integer or strings, but also common message types for navigation, geometry or sensors. These standardized message types have an advantage when it comes to reusing code. Documentation of the different message types is available on the ROS wiki website. If the same message type is used for several subprojects, those subprojects can easily communicate with each other. In addition to that, new message types can be created and implemented if the standard message types are not suitable for a project.

Some of the most important message types for this work are Joint State, Pose Stamped, Twist and Transform Stamped. A message of the type Joint State includes a standard header with an ID number, the time in seconds and nanoseconds since the beginning of the UNIX time and the frame ID that this message belongs to. Also, the name of the joint, as well as its current position, velocity and effort that is applied in the joint is transmitted. If one of the information is not available, the array can also be left empty. Pose Stamped is a message type with a standard header that includes the pose (position and orientation) of an object. Twist is a geometry message that expresses the velocity of an object in linear and angular direction. Both linear and angular velocity are presented in vectors of the size three with one entity for each axis. Transform Stamped is a message with a standard header which expresses the transformation between a child and a parent coordinate system. It is mostly used by the tf package and is an important input for most localization and mapping algorithms. A tf message consists of an array of Transform Stamped messages. It represents a so-called tf tree, which should include all necessary transformations of a setup. Necessary transformations can be from a map to the base frame of a robot or from a base frame to a specific subcomponent of the robot. The tf message keeps track of all coordinate frames and the transformations between them over time. It lets the user access that information for previous points in time as well as for the current time. Tf can operate in a distributed system and therefore makes the stored information about all coordinate frames of a robot available to all ROS-based components in all computers in a system.

On the computation graph level, different processes in the peer-to-peer network are processing and providing data together. In a peer-to-peer network, nodes have the capability to act as a server and a client at the same time. It is seen as the opposite of a Client and Server architecture, in which nodes can only act as a server or as a client. The concepts within the computation graph level are for example the ROS master, nodes, messages, topics and bags. The ROS master provides different services and information to the nodes. For example, it provides services for naming and registration and keeps track of all publishers and subscribers in a ROS system. The role of the ROS master is to enable the ROS nodes to locate one another. Most often, the ROS master is loaded via the roscore command. Nodes perform computations, as well as publish and subscribe to topics. The data that the nodes publish and subscribe to the topics is structured as messages.

An example of several nodes that publish and subscribe to different topics can be seen in below figure. The nodes are indicated with an ellipse shape while the topics are presented in a rectangular shape. The arrows between the nodes and topics represent publishing and subscribing actions. The direction of the arrow defines, if a node was publishing or subscribing to a topic. If the arrow points in the direction of the topic, the message is being published to that topic. On the other hand, if the arrow points in the direction of the node, the node is subscribing to that topic. In this example, there can be seen three nodes and three topics. The robot_state_publisher is publishing a message to tf_static and the rosbridge_websocket is publishing to scan. All three nodes are publishing to the tf topic. In addition to that, the slam_gmapping node is subscribing to all available topics. Therefore, it publishes and subscribes to the tf topic at the same time.

This is possible and needed in this case, as the slam_gmapping node is updating the tf message within the tf topic. It needs to be mentioned that nodes can only communicate with each other if the ROS master is running. In figure Example of nodes that publish and subscribe to different topics in ROS. Messages can be saved and played back with so-called ROS bags. The ROS bag format is a logging format which can be used for storing ROS messages in files. The current version of ROS bags, 2.0, contains features that provide the possibility to compress the bag and store the messages by connection to improve the possibilities for republishing the messages from the bag. This way of recording message data can be used for example for evaluating purposes or repeating the same control inputs that have been recorded before. Tools like rosbag and rqt_bag enable the usage and manipulation of the bags. The third level of concept, the ROS community level, enables different communities of ROS users and researchers to exchange their software and their knowledge with each other. The resources in the community level are for example distributions, repositories, the ROS wiki and ROS answers. Distributions are a collection of stacks that can be used for installing ROS. They include sets of software and are available for different platforms and in different memory sizes including different sets of packages. In repositories, own robot software components can be developed and released under open-source licenses by different institutions. Those repositories can then be used, improved and further developed by other institutions and users. The ROS wiki and answers provide the possibility for users to share their knowledge and cooperate to solve problems. The usage of ROS and its community have been constantly growing in the past years. In the annual metrics report of July 2018, the open robotics team lists 2,204,869 page views for July 2018 with an annual growth of 21%. It is stated that the number of different robot types available for the community with ROS drivers is constantly rising and was at approximately 130 in July 2018. The growing community and the resulting rise of research and contributions make ROS a desirable tool to use for robotics research and robot development.

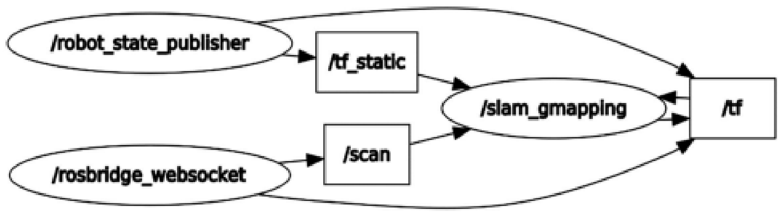


Figure 1. Example of nodes that publish and subscribe to different topics in ROS.

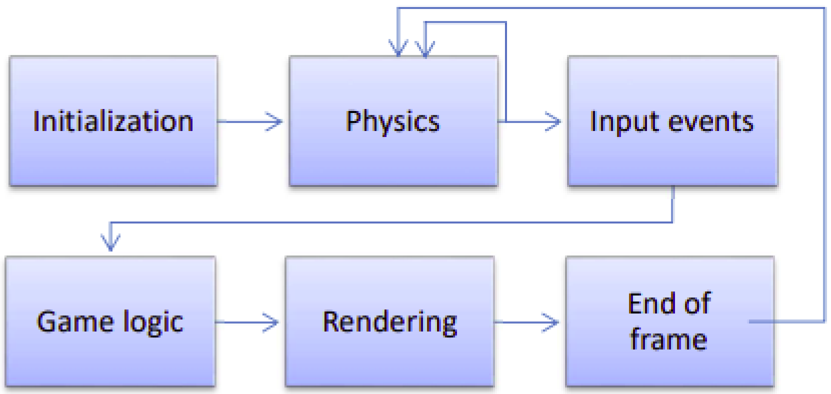
Unity

Unity is a game engine that was developed by Unity Technologies. It was created in 2005 and has since then become a widely used engine for the development of games. A project in Unity can consist of multiple scenes. Traditionally, those scenes are game levels or menus. They give the opportunity to structure and debug the project in a logical and modular manner. Scenes consist of various game objects, which themselves contain different types of components. The components can define the functionality and behavior of game objects. For example, components can be joints, colliders or renderers. Game objects can be structured in a hierarchy and thereby be in a parent-child relationship to other game objects. In that way, a single robot can be presented by one game object parent which contains multiple child game objects. Unity provides the user the ability to use an object-oriented framework for scripting in C# and Unity Script. The latter is specifically designed for Unity and modelled after JavaScript. In addition to those two programming languages, other .Net languages can be used with Unity under certain prerequisites. There are two types of object classes in Unity: Runtime and Editor. The Runtime class contains scripts that are used as new types of game components. One or multiple of these scripts can be attached to one or multiple game objects and control their behavior during game play. Scripts of the Editor classes on the other hand can add menu items to the default Unity menu system.

With such scripts, for example new types of game objects can be added to the Unity editor. Any item that can be used within a project, no matter if it is a scene, a game object or a component, can be turned into an asset. An asset is a representation of that item. It can be either imported from outside of Unity or be made inside of Unity. Files from common CAD software can be imported into Unity and be made an asset. Assets can also be purchased or downloaded for free from Unity's Asset Store. Assets can be reused within one project and also be copied into other projects. Most actions in the simulation in Unity run at a variable frame rate, while the physics calculations run at a fixed timestep.

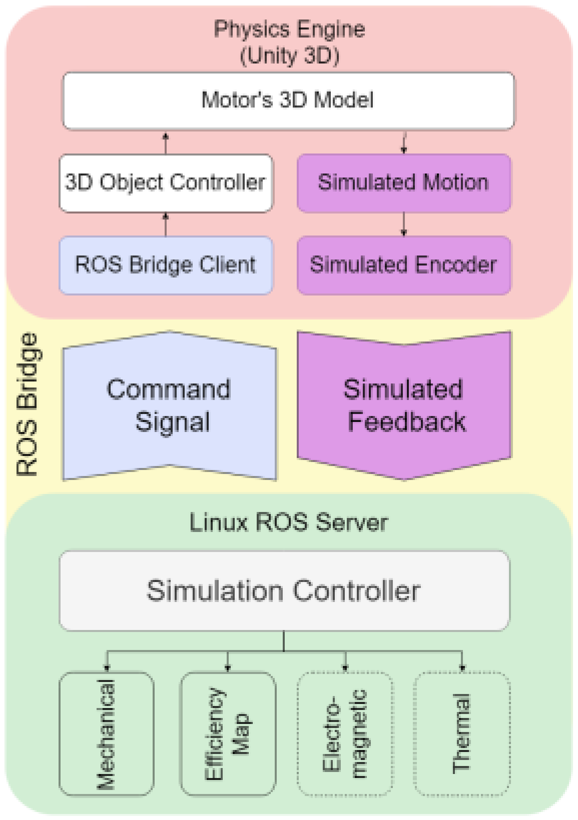
The ordering and repetition of event functions during one simulation can be seen in Figure. The simulation starts with the initialization of all scripts and events. Afterwards, the physics calculations are executed. These calculations are called fixed update and are based on the fixed timestep. That means that if the fixed time step is less than the actual frame update time, the physics calculations can be repeated more than one time per frame. If the fixed time step is larger than the actual frame update time, it can also happen that the physics calculations are not executed during one frame. After the physics calculations, the input events from the user are processed.

For example, scripts that are triggered by the user with a mouse or keyboard input. When those inputs are processed, the game logic is executed. Afterwards, the scene and GUI (Graphical User Interface) are rendered. As soon as this is finished, the end of the frame is reached. The end of the frame triggers another frame which starts with the physics calculations, if those need to be computed. The physics calculations in Unity are, as common for simulators, done by physics engines that are integrated in the simulation software. There are various physics engines available and many of them evolved from different backgrounds. Because of this, they vary in used approaches and algorithms.



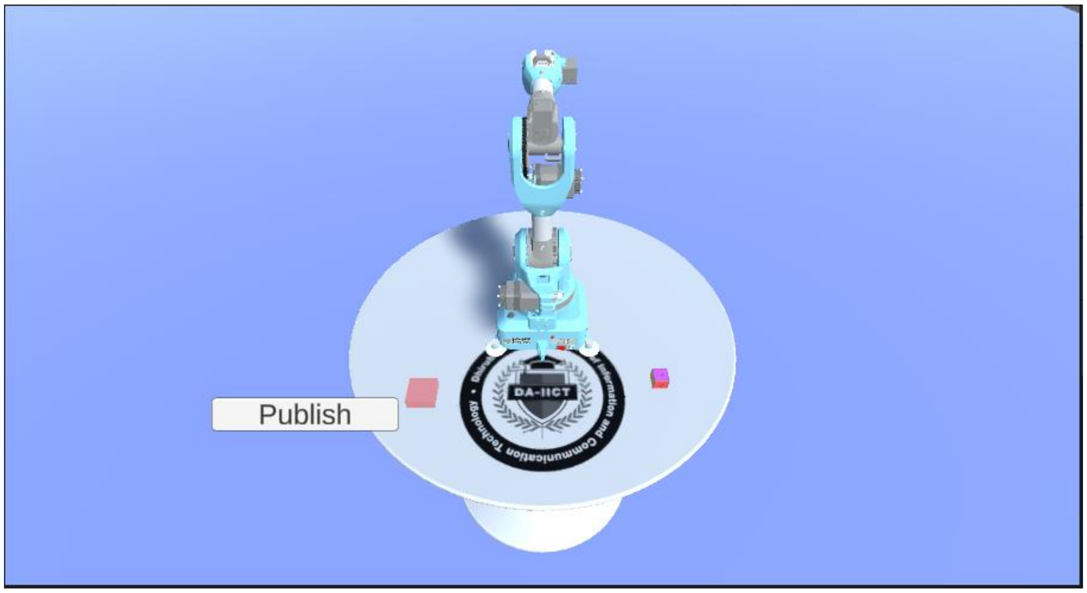
Case Study

For this report, we are going to study the latency for the robot control. Below is the near architecture of our designed software.



Underlying Architecture

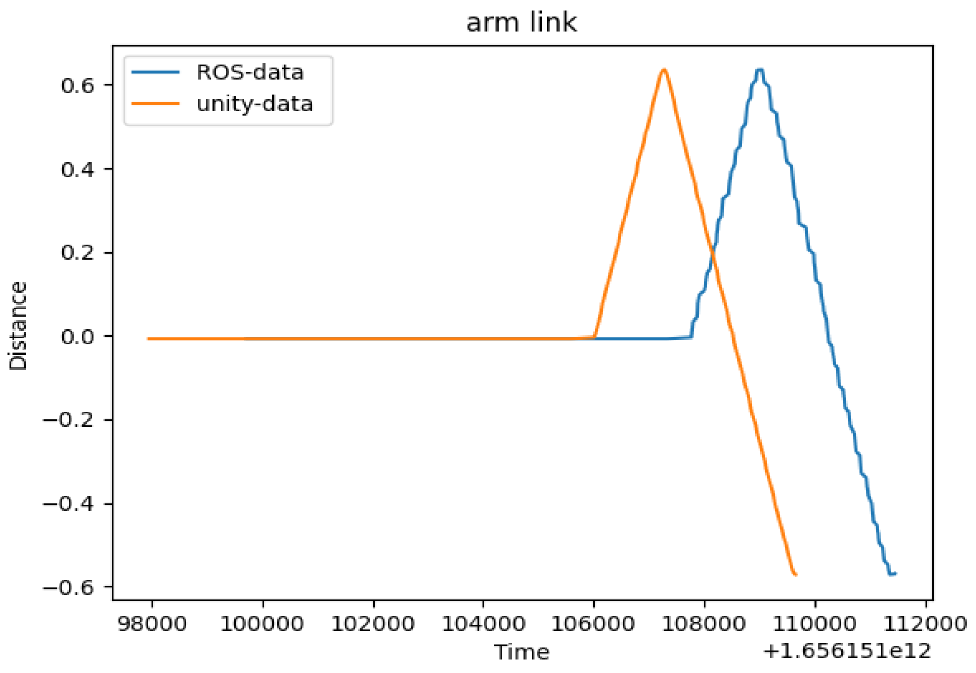
Robot Model:

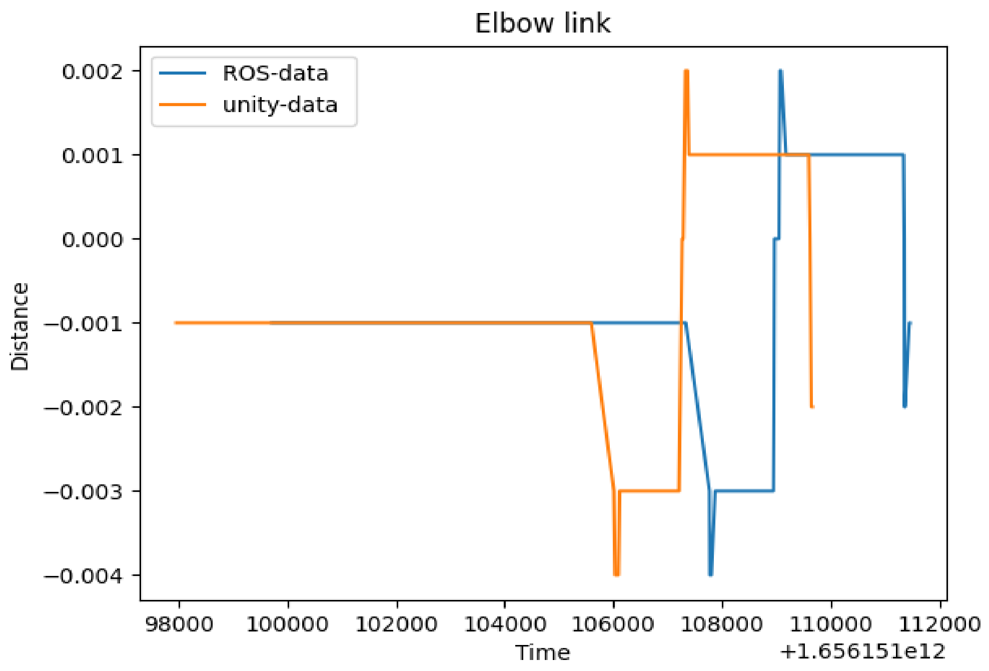

Publish
DA-IICT
Dhirubhai Ambani Institute of Information and Communication Technology

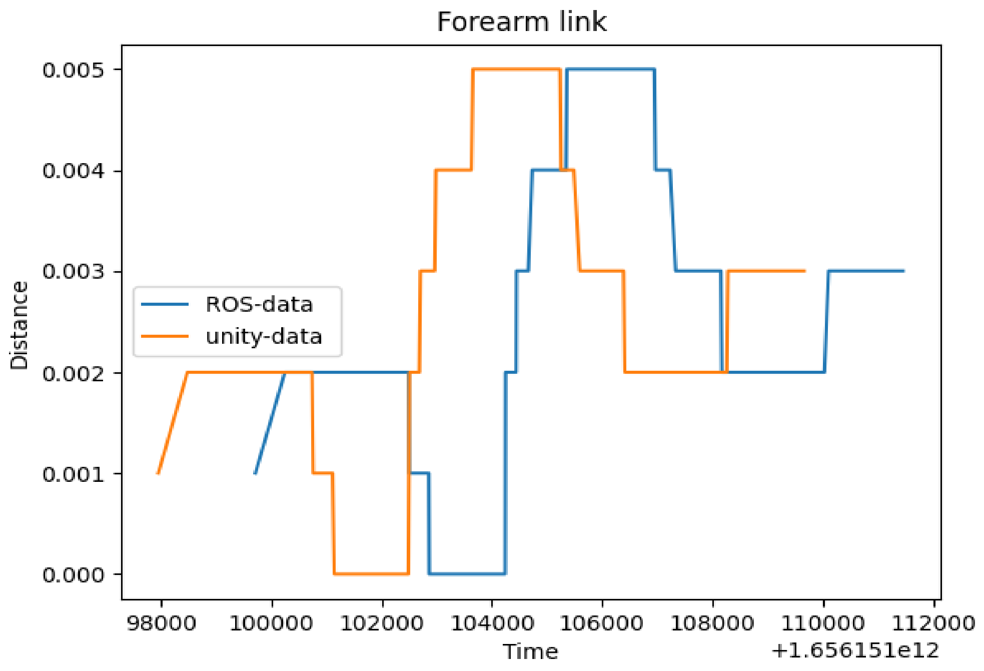
Our Niryo robot (robotic arm) as shown in the image is used for our specific case study. We are going to control it. Then we will study latency between ROS sub-system and Unity sub system. There are many kinds of operations that can cause delay, for example delay between keyboard input and robot movement in the software, delay between data gathering from unity to writing data to a file, passing the data over ROS bridge, reading data and writing it to a file on ROS side. However, our guess is that the delay which is in milli seconds in our case study is majorly causing due to network transportation.

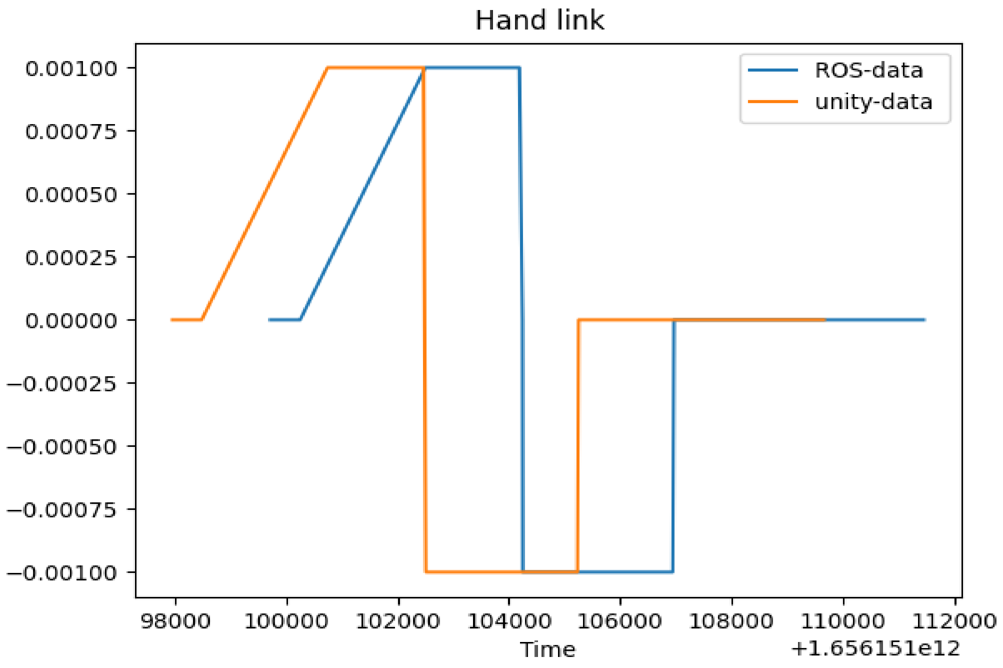
Niryo robot has multiple joints namely shoulder link, arm link, elbow link, forearm link, wrist link and hand link. We will control this robot links through keyboard and will study latency through graph. For the scope of this report, we will study single way latency. Here, we have used virtual network over a real network for our study. Below are the data plots of various links. Given data is a unidirectional latency.

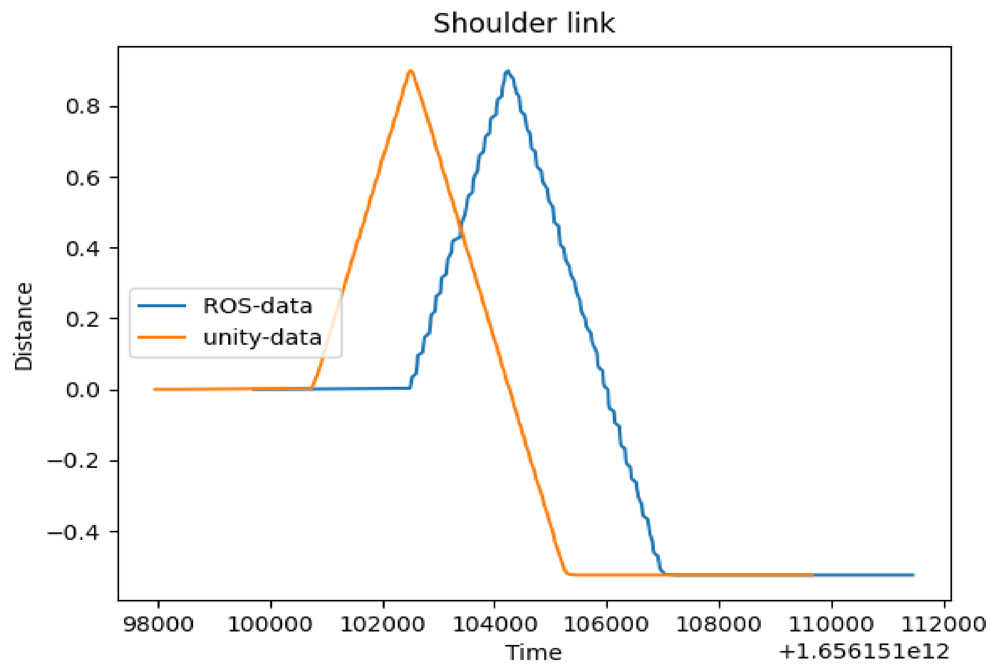
Unidirectional Latency Plot

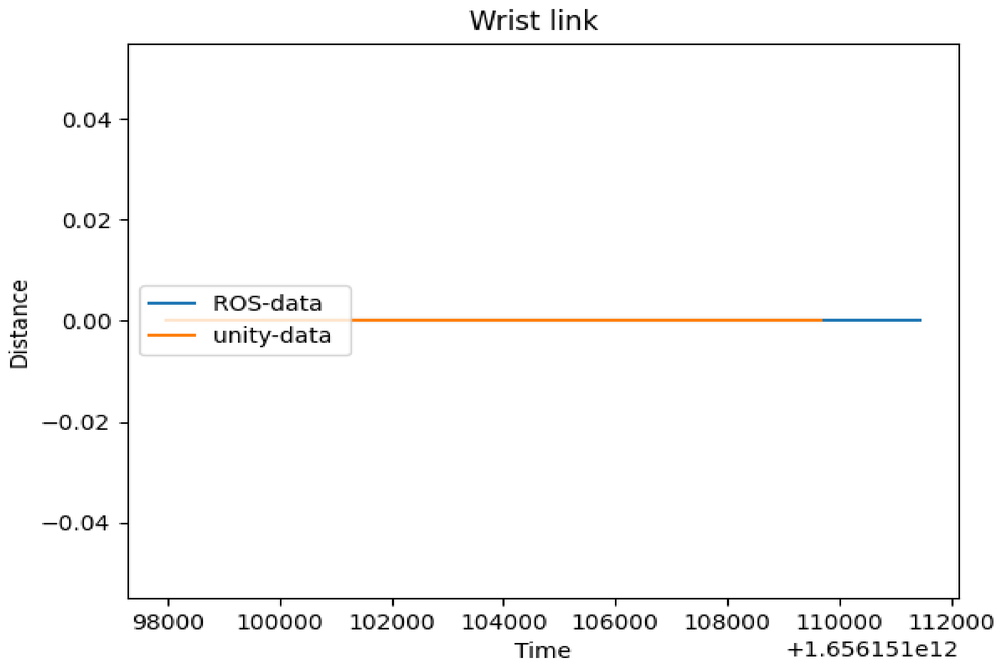



Elbow link
ROS-data
unity-data
Distance
0.002
0.001
0.000
−0.001
−0.002
−0.003
−0.004
98000
100000
102000
104000
106000
108000
110000
112000
Time
+1.656151e12


Forearm link
ROS-data
unity-data
Distance
0.005
0.004
0.003
0.002
0.001
0.000
98000
100000
102000
104000
106000
108000
110000
112000
Time
+1.656151e12


Hand link
ROS-data
unity-data
0.00100
0.00075
0.00050
0.00025
0.00000
−0.00025
−0.00050
−0.00075
−0.00100
98000
100000
102000
104000
106000
108000
110000
112000
Time
+1.656151e12


Shoulder link
Distance
Time
+1.656151e12
ROS-data
unity-data
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
98000
100000
102000
104000
106000
108000
110000
112000


Wrist link
Distance
Time
+1.656151e12
ROS-data
unity-data
0.04
0.02
0.00
−0.02
−0.04
98000
100000
102000
104000
106000
108000
110000
112000

Latency Study

Here we have used Unity platform on one operating system and deployed ROS through docker on another operating system. And both the system is communicating through ROS bridge. And ROS bridge is implemented on virtual network (Hamachi in our case).

There are many causes that can an affect latency like keyboard commands, internal processing of difference software, managing file systems and many more. But from our experience and study we can say that these causes will just lead to delay of micro or nano seconds. However, the major cause is network delay which leads to latency having range of milliseconds.

Solution to Latency

1) **Port forwarding** – One solution is port forwarding setup on routers. Currently for the scope of the report entire network setup is on virtual network which will be obviously slower than the real network. To solve this, we can implement port forwarding setup on the rou ters which are connected to the PCs. This will cause into faster data propagation from both the side.
2) **Using UDP** – Another solution is transferring the transport layer protocol from TCP to UDP. As UDP is faster than TCP. We will have faster network setup than the current one, as current network setup is using TCP for the data transmission.

C# Script

```
using System;
using System.IO;
using System.Collections;
using System.Collections.Generic;
using RosMessageTypes.Geometry;
using RosMessageTypes.NiryoMoveit;
using Unity.Robotics.ROSTCPConnector;
using Unity.Robotics.ROSTCPConnector.ROSGeometry;
using Unity.Robotics.UrdfImporter;
using UnityEngine;

public class SourceDestinationPublisher : MonoBehaviour
{
    const int k_NumRobotJoints = 6;
   public static readonly string[] LinkNames =
        { "world/base_link/shoulder_link", "/arm_link", "/elbow_link", "/forearm_link", "/wrist_link",
"/hand_link" }
    // Variables required for ROS communication
    [SerializeField]
    string m_TopicName = "/niryo_joints";
    double[] jointsArray = new double[6];
    [SerializeField]
    GameObject m_NiryoOne;
    [SerializeField]
    GameObject m_Target;
    [SerializeField]
    GameObject m_TargetPlacement;
    readonly Quaternion m_PickOrientation = Quaternion.Euler(90, 90, 0);

    // Robot Joints
    UrdfJointRevolute[] m_JointArticulationBodies;
```

```
// ROS Connector
    ROSConnection m_Ros;
    string filePath;
    StreamWriter writer;
    void Start()
    {
        // Get ROS connection static instance
        m_Ros = ROSConnection.GetOrCreateInstance();
        m_Ros.RegisterPublisher<NiryoMoveitJointsMsg>(m_TopicName);

        m_JointArticulationBodies = new UrdfJointRevolute[k_NumRobotJoints];

        var linkName = string.Empty;
        for (var i = 0; i < k_NumRobotJoints; i++)
        {



            linkName += LinkNames[i];
            m_JointArticulationBodies[i]                                                                                  =
m_NiryoOne.transform.Find(linkName).GetComponent<UrdfJointRevolute>();
        }

        filePath = "data/data.txt";
        File.Create(filePath).Close();
        writer = new StreamWriter(filePath, true);
    }
    // run every second
    void Update(){

     if(jointsArray[0] != Math.Round(m_JointArticulationBodies[0].GetPosition(),3) ||
        jointsArray[1] != Math.Round(m_JointArticulationBodies[1].GetPosition(),3) ||
        jointsArray[2] != Math.Round(m_JointArticulationBodies[2].GetPosition(),3) ||
        jointsArray[3] != Math.Round(m_JointArticulationBodies[3].GetPosition(),3) ||
        jointsArray[4] != Math.Round(m_JointArticulationBodies[4].GetPosition(),3) ||
        jointsArray[5] != Math.Round(m_JointArticulationBodies[5].GetPosition(),3) )
```

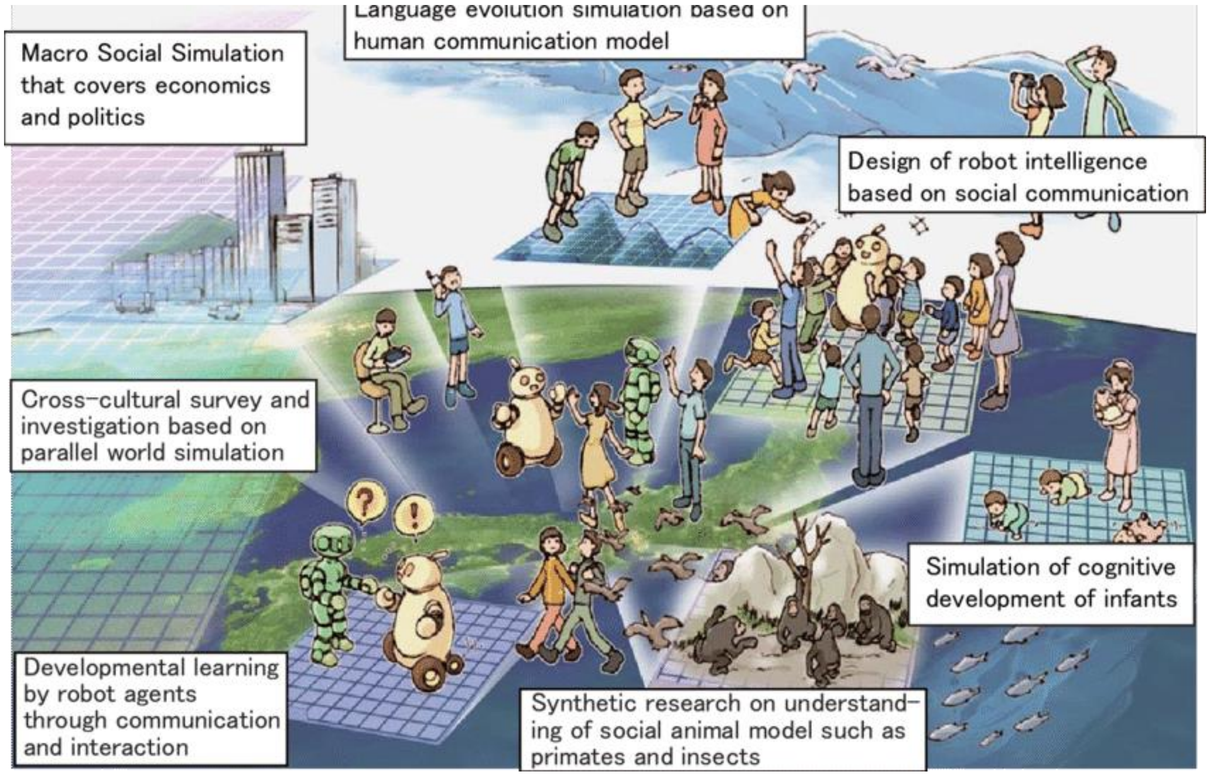
```
{
        var sourceDestinationMessage = new NiryoMoveitJointsMsg();
        long unixTimestamp = new DateTimeOffset(DateTime.UtcNow).ToUnixTimeMilliseconds();
        string s = "";
        for (var i = 0; i < k_NumRobotJoints; i++)
        {
            sourceDestinationMessage.joints[i] = Math.Round(m_JointArticulationBodies[i].GetPosition(),3);
            jointsArray[i] = Math.Round(m_JointArticulationBodies[i].GetPosition(),3);
            s = s + jointsArray[i] + ",";
        }
        s = s + unixTimestamp;

        writer.WriteLine(s);

        m_Ros.Publish(m_TopicName, sourceDestinationMessage);
    }



    }

    public void Publish()
    {
        var sourceDestinationMessage = new NiryoMoveitJointsMsg();

        for (var i = 0; i < k_NumRobotJoints; i++)
        {

            sourceDestinationMessage.joints[i] = m_JointArticulationBodies[i].GetPosition();
        }
        // Pick Pose
        sourceDestinationMessage.pick_pose = new PoseMsg
        {
            position = m_Target.transform.position.To<FLU>(),
            orientation = Quaternion.Euler(90, m_Target.transform.eulerAngles.y, 0).To<FLU>()
        };
        // Place Pose
        sourceDestinationMessage.place_pose = new PoseMsg
        {
            position = m_TargetPlacement.transform.position.To<FLU>(),
            orientation = m_PickOrientation.To<FLU>()
        };

        // Finally send the message to server_endpoint.py running in ROS
        m_Ros.Publish(m_TopicName, sourceDestinationMessage);
    }
    void OnDestroy(){
        writer.Close();
    }
}
```

ROS Python Scrip

```
#!/usr/bin/env python
"""
    Subscribes to SourceDestination topic.
    Uses MoveIt to compute a trajectory from the target to the destination.
    Trajectory is then published to PickAndPlaceTrajectory topic.
"""
import rospy
import time
from niryo_moveit.msg import NiryoMoveitJoints, NiryoTrajectory
from moveit_msgs.msg import RobotTrajectory

def callback(info):
    data = info.joints
    ts = int(time.time()*1000)
    s =
str(data[0])+","+str(data[1])+","+str(data[2])+","+str(data[3])+","+str(data[4])+","+str(data[5])+","+str(ts)
    f =open("data.txt",'a')
    f.write(s)
    f.write('\n')
    f.close()
    rospy.loginfo(rospy.get_caller_id() + "I heard:\n%s", info)

def listener():
    rospy.init_node('Trajectory_Subscriber', anonymous=True)
    rospy.Subscriber("/niryo_joints", NiryoMoveitJoints, callback)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

if __name__ == '__main__':
    listener()
```

Future Scopes

A real human (test subject) can log into the virtual world through general VR devices to interact with the virtual robot in the virtual world. You can apply the original software, which is used to control the real robot based on ROS, to the virtual robot without any modification.


Language evolution simulation based on human communication model
Macro Social Simulation that covers economics and politics
Design of robot intelligence based on social communication
Cross-cultural survey and investigation based on parallel world simulation
Simulation of cognitive development of infants
Developmental learning by robot agents through communication and interaction
Synthetic research on understand-ing of social animal model such as primates and insects

The above image shows a multi-modal system architecture. This system can be used to command robots and perform several tasks which cannot be done by real humans at a time. And further cloud technologies can be integrated to study the modal interactions and can ultimately be made intelligent autonomous systems.