Universitat de Girona
Escola Politècnica Superior

Grau en Enginyeria Informàtica

PROJECTE FINAL DE GRAU

Comunicació ROS en xarxes poc fiables i mòdems òptics subaquàtics

Autor:
Enric Pagès Agustí

Tutors:
Narcís Palomeras Rovira
Roger Pi Roig

MEMÒRIA

Convocatòria:
Juny 2023

Departament :
Arquitectura i Tecnologia de computadors

Projecte: Projecte Final de Grau
Document: Memòria
Títol: Comunicació ROS en xarxes poc fiables i mòdems òptics sub-aquàtics
Autor: Enric Pagès Agustí
Data: Juny 2023

Estudi:
Grau en Enginyeria Informàtica
Universitat de Girona

Supervisor 1:
Narcís Palomeras Rovira
Universitat de Girona
Email: narcis.palomeras@udg.edu

Supervisor 2:
Roger Pi Roig
Universitat de Girona
Email: roger.piroig@gmail.com

Índex

1 Introducció, motivacions, propòsit i objectius del projecte **1**
1.1 Introducció 1
1.2 Motivacions i propòsit 2
1.3 Objectius principals 2

2 Estudis de viabilitat **3**
2.1 Viabilitat econòmica 3
2.2 Viabilitat tècnica 3
2.3 Viabilitat legal 4

3 Metodologia i planificació **5**
3.1 Metodologia 5
3.1.1 Scrum 5
3.1.2 Aplicació de Scrum en el projecte 6
3.2 Planificació 6

4 Conceptes Prèvis **7**
4.1 Mòdems òptics LUMA 7
4.2 CIRS i els submarins Girona 8
4.3 ROS 8
4.4 Protocols sobre IP 9

5 Requisits del sistema **11**
5.1 Requisits funcionals 11
5.2 Requisits no funcionals 11

6 Estudis i Descisions **12**
6.1 Python i C++ 12
6.2 OpenCV 12
6.3 ROS_RTSP 12
6.4 Protocols sobre IP 13
6.4.1 Protocol TCP 13
6.4.2 Protocol UDP 14
6.4.3 Protocol RTSP 14

7 Anàlisi i disseny del sistema **16**
7.1 Comunicació estàndard de ROS 16
7.2 Enviament de paquets en ROS 17
7.3 Aplicació particular 20
7.4 Concepte rere el ROS2ROS 20

8 Implementació i Desenvolupament **23**

8.1 Primers passos 23
8.2 Primeres proves locals 23
8.2.1 Testatge de *topics* 23
8.2.2 Testatge de services 25
8.3 Instal·lació LUMA 25
8.4 Proves de banda d'amplada LUMA 28
8.5 Transmissió de vídeo 29
8.5.1 Enviament d'imatge contínua 29
8.5.2 Compressió temporal 29
8.5.3 Compressió Theora 30
8.5.4 RTSP 31

9 Implementació i Resultats **32**
9.1 Enviament de missatges 32
9.1.1 Simulació en màquines virtuals 32
9.1.2 Primeres proves en els mòdems 33
9.2 Transmissió de vídeo 33
9.2.1 Saturament de la xarxa 33
9.2.2 Augment de receptors 35
9.2.3 Límit de l'amplada de banda emprat per la màquina 36
9.3 Eina Iperf 37

10 Implementació final **41**
10.1 Paquet ROS2ROS 41
10.2 Codi remitent de *topics* (*sender.py*) 41
10.2.1 Codi principal 41
10.2.2 Fil d'execució UDP 44
10.2.3 Callback UDP 44
10.2.4 Fil d'execució TCP 45
10.2.5 Callback TCP 46
10.3 Codi receptor de *topics* (*receiver.py*) 46
10.3.1 Codi principal 46
10.3.2 Fil d'execució UDP 46
10.3.3 Fil d'execució TCP 47
10.3.4 Fil d'execució RTSP 48
10.4 Enviament RTSP 50
10.5 Codi client de serveis per TCP (*tcp_client*) 50
10.5.1 Codi principal 50
10.5.2 Callback 51
10.6 Codi servidor de serveis per TCP (*tcp_client*) 51
10.6.1 Codi principal 51
10.6.2 Fil d'execució del servidor 52
10.7 Arxius de configuració 53
10.7.1 Configuració de la retransmissió de *topics* 53
10.7.2 Configuració de la retransmissió de serveis 54
10.8 Fitxers d'execució 55
10.8.1 Fitxers *.launch* 55
10.8.2 Fitexer d'execució de la comunicació per *topics* 55
10.8.3 Fitexer d'execució de la comunicació per serveis 56
10.8.4 Fitexer d'execució de l'aplicació d'enviament de paquets RTSP 56

11 Conclusions **58**

12 Treball Futur **59**
12.1 Un únic YAML . . . 59
12.2 Millores de codi . . . 59
12.3 Creació instantània de subscriptors . . . 59

A ros2ros **62**
A.1 Description . . . 62
A.2 Dependencies . . . 62
A.3 Usage . . . 62
A.4 YAML Setup . . . 62
A.4.1 config_receiver.yaml . . . 62
A.4.2 config_sender.yaml . . . 63
A.4.3 stream_setup.yaml . . . 64
A.4.4 config*tcp*service.yaml . . . 64
A.4.5 config*tcp*client.yaml . . . 65

Capítol 1

Introducció, motivacions, propòsit i objectius del projecte

1.1 Introducció

En el món actual de la robòtica és pràcticament un estàndard l'ús de la infraestructura proporcionada per ROS, un programari que gestiona diferents components d'un sistema robòtic mitjançant execucions aïllades anomenades nodes. Aquests nodes comparteixen informació a través de canals públics anomenats *topics*, els quals funcionen per connexions de xarxa, governades per un node principal anomenada *roscore*, utilitzant protocols d'enviament segurs però costosos d'amplada de banda.

En els àmbits on se sol operar, la connexió entre els nodes és prou fiable com per retransmetre qualsevol mena de dades, a través de cables de xarxa o connexió sense fils, ja sigui text, imatge, vídeo, núvols de punts, etc; però en el món subaquàtic no és tan senzill. En aquest, fins ara, era imperatiu l'ús de xarxes únicament formades per cables, perquè l'ús d'ones de ràdio, el recurs més típic per comunicar-se sense fils, no és viable dins de l'aigua, ja que aquestes ones perden potència dramàticament amb molt poca distància.

Recentment, s'ha posat el focus en la comunicació subaquàtica fent servir mòdems òptics, els quals són capaços de comunicar-se modulant llum, aconseguint molta més amplada de banda i baixa latència que la comunicació acústica, la qual és l'alternativa sense fils convencional a sota aigua. Tot i que la distància d'ús és limitada (< 50 m), aquest tipus de mòdems obren la porta a un gran nombre d'aplicacions, com monitorar i teleoperar el robot en temps real en infraestructures submarines. Tot i això, la seva connexió no és del tot fiable per culpa del medi aquàtic en el qual treballen.

Aquest projecte té com a objectiu principal idear i desenvolupar una solució genèrica per poder habilitar i maximitzar l'eficiència de les comunicacions entre nodes de ROS en xarxes que disposen de poca amplada de banda i una pèrdua alta de paquets, aplicant aquesta solució als mòdems òptics Luma X, uns mòdems òptics dissenyats per a la robòtica submarina dels quals disposa el CIRS.

Així doncs, en aquest treball es farà un estudi del mòdem òptic, entenent-ne el funcionament i identificant-ne com es pot maximitzar la qualitat de la connexió, a més de planejar com es farà la connexió entre aquests, i dissenyar les estratègies i protocols a seguir per cada situació.

1.2 Motivacions i propòsit

La motivació principal per dur a terme aquest projecte és la motivació personal d'innovar en un camp de recerca actual, com pot ser la robòtica submarina, i interactuar amb noves tecnologies, treballant amb l'ajuda d'una empresa local.

A més a més, aquest projecte en concret és interessant perquè dona l'oportunitat d'aplicar coneixements adquirits cursant la carrera universitària, com les xarxes IP, o la robòtica gestionada per ROS, en un hardware real que, a més a més, és nou al mercat i té un gran potencial en l'àmbit de la recerca submarina.

1.3 Objectius principals

Com esmentat al apartat 1.1, l'objectiu d'aquest treball és proposar i desenvolupar una solució genèrica per poder comunicar programes (nodes) de ROS en xarxes poc fiables i amb poca amplada de banda. Per fer-ho, es pretén usar els mòdems òptics Luma X com a enllaç no fiable.

Per tant, els objectius desglossats serien els següents:

- Desenvolupar una aplicació genèrica que permeti la comunicació bidireccional de qualsevol mena d'aplicació distrïbuida de ROS a través de xarxes amb poca amplada de banda i poc fiables.
- Aplicar la solució desenvolupada als mòdems òptics LUMA i adaptar el codi a les seves necessitats.
- Idear i implementar solucions per millorar l'eficiència del codi i dels mòdems.
- Aconseguir enviar entre nodes de ROS diferents tipus de missatges de manera eficient, com ara missatges basats en text, com poden ser diagnòstics de l'estat d'un robot o odometria, imatge, i vídeo.

Capítol 2

Estudis de viabilitat

Si es tenen en compte les hores invertides per l'estudiant com a hores realitzades per un programador júnior, mentre que les hores de tutoria es compten com hores proporcionades per programadors sèniors supervisant el projecte, el còmput total seria el vist a la taula 2.1.

2.1 Viabilitat econòmica

Concepte	Quantitat	Cost unitari	Total
Hores programador júnior	250	14€	2500€
Hores programador sènior	10	25€	250€
Mòdem òptics Luma X	2	5000	10000
Cables connexió Luma	2	480€	960€
Total			**13710€**

TAULA 2.1: Cost estimat del projecte

En aquest càlcul s'obvia els costs de manteniment de les instal·lacions i l'equipament addicional usat. Tot el software utilitzat és de codi obert o d'ús gratuït, així que no es té en compte en el còmput.

Els preus dels salaris s'han extret del portal Talent.com [1], mentre que els preus del material dels Luma prové de l'empresa Hydromea[2], que els manufactura.

2.2 Viabilitat tècnica

El material necessari per dur a terme el projecte és:

- Dos ordinadors amb ROS instal·lat.
- Els mòdems òptics Luma X, i el cablejat necessari per endollar-los als ordinadors.
- Un contenidor d'aigua prou gran per recrear l'entorn de treball dels mòdems.

Tot aquest material és proporcionat pel centre d'investigació en Ròbotica Submarina (CIRS) de la UdG.

2.3 Viabilitat legal

Tots els paquets i llibreries que s'utilitzen per desenvolupar el codi són de tipus *open source*. És a dir, són projectes on el seu codi font és disponible al públic per consultar, modificar, i distribuir sense cap mena de restricció.

El codi desenvolupat també tindrà una llicencia de codi obert per desambiguar qualsevol incidència al voltant de la seva legalitat.

Per tant, el projecte és viable legalment, fins i tot si se'n distribuís el codi posteriorment.

Capítol 3

Metodologia i planificació

3.1 Metodologia

La metodologia que s'ha seguit en la realització d'aquest projecte ha estat la del marc de desenvolupament *Scrum*.

3.1.1 Scrum

Scrum és un mètode de gestió de projectes complexos apilament usat en la indústria del desenvolupament de software. Aquest mètode proporciona un enfocament estructurat al plantejament, l'organització, i la implementació i desenvolupament de productes o serveis seguint un format iteratiu i incremental.

Un projecte gestionat amb Scrum és desenvolupat de manera iterativa, dividint les tasques a realitzar en períodes de temps anomenats Sprints. Per cada Sprint, es planeja quin objectiu es vol assolir, es treballa en aquest objectiu durant el període establert, i, finalment, s'avaluen els resultats un cop finalitzat el període. En acabar un Sprint, es fa retrospectiva del que s'ha aconseguit i es plantegen els passos a seguir en el següent Sprint si no s'ha acabat el treball. Es repeteix aquest cicle fins a la finalització del projecte.

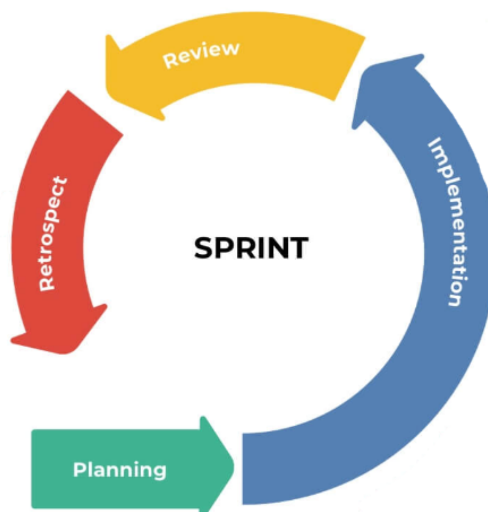

FIGURA 3.1: Estructura d'un Sprint de Scrum

3.1.2 Aplicació de Scrum en el projecte

Scrum és pensat per a feines en les quals hi treballen diverses persones, assignant rols com Scrum Master, qui dirigeix el desenvolupament, o desenvolupador de codi, o propietari del producte. Com que aquest projecte en qüestió ha estat dut a terme de forma individual, s'ha modificat el marc de desenvolupament per acomodar aquestes necessitats, prescindint d'aspectes orientats a les tasques de grup, com pot ser assignar rols.

La metodologia que s'ha utilitzat ha consistit en Sprints setmanals o bisetmanals, on s'han seguit els següents passos:

1. Planificar les tasques a fer durant l'Sprint.
2. Dur a terme les tasques acordades.
3. Avaluar individualment els resultats, modificant els aspectes no satisfactoris amb el plantejament de l'Sprint si és possible.
4. Fer retrospectiva del progrés assolit comentant els resultats amb els tutors del projecte.

3.2 Planificació

El projecte va començar seguint un plantejament inicial pels primers Sprints, i desrpés es va anar evolucionant a mesura que anava avançant el treball.

La planificació de tot el projecte ha estat la següent:

1. Cerca d'informació de com gestionar la comunicació d'una xarxa amb poca amplada de banda i poca fiabilitat.
2. Implementació una solució per al framework ROS.
3. Testatge del funcionament en màquines virtuals amb una xarxa estable.
4. Instal·lació i comprovació de connexió d'un canal de comunicació poc fiable i amb pèrdues.
5. Testatge del funcionament del codi en màquines reals mitjançant el canal instal·lat.
6. Desenvolupament de millores segons les observacions de les proves.

Capítol 4

Conceptes Prèvis

4.1 Mòdems òptics LUMA

Els mòdems òptics Luma X, desenvolupats per l'empresa suïssa Hydromea [2], especialitzada en robòtica submarina, són innovadors dispositius dissenyats per a la comunicació subaquàtica i són el focus principal d'aquest projecte.

En el món de la investigació subaquàtica, els robots teleoperats pràcticament sempre estan connectats a través d'un cable per permetre la comunicació amb l'operari, ja que, fins ara, no hi havia mètodes viables de comunicació sense fils que permetessin un trànsit de dades suficient per a aquestes aplicacións. Per culpa del medi aquàtic, la comunicació sense fils és una tasca força més complicada que en la superfície, perquè les ones de ràdio que típicament s'utilitzen per a aquesta mena de comunicacions perden potència extremadament ràpid sota l'aigua i no són capaces propagar-se a cap distància significativa.

Les úniques opcions viables per enviar dades a distància en aquests casos és, o bé polsos acústics (com ara els *sonars* que usen els submarins militars), o bé la modulació de llum. Com que el *sonar* té una latència que augmenta ràpidament amb la distància, ja que la seva velocitat de propagació està limitada per la velocitat del so a l'aigua, i una amplada de banda molt reduït (en l'ordre de pocs kbps), els enginyers d'Hydromea van optar per crear uns mòdems de comunicació a través de la modulació de llum. Tot i tenir desavantatges, com un rang dramàticament menor, o més possibilitat de distorsió, per a l'aplicació de la robòtica submarina és crucial tenir una latència reduïda per poder dur a terme accions en temps real, com ara controlar un submarí o rebre'n informació com imatges de la seva càmera, odometria, o diagnòstics, fent els mòdems òptics actualment la millor opció del mercat.

Els mòdems òptics Luma X són l'última invenció de l'empresa suïssa, els quals fan servir missatges en forma de polsos de llum per solucionar el problema de la comunicació sense fils. Una parella d'aquests mòdems, connectats cadascun a una màquina diferent, permeten connectar dues interfícies de xarxa transparentment.

Bàsicament, cada Luma X consisteix en un emissor de llum LED i un receptor de llum, els quals s'encarreguen d'enviar i rebre dades. Quan dos mòdems estan enfocats l'un amb l'altre, un envia informació modulant llum, mentre que l'altre capta la llum amb els seus receptors i en descodifica la informació transmesa. Els mòdems poden emetre i captar llum a la vegada, així que la comunicació és bidireccional.

En una xarxa corrent els paquets d'informació no són res més que uns i zeros, que en els cables de coure s'identifiquen com a pas d'electricitat (uns) o talls del flux (zeros). Els Luma X funcionen de la mateixa manera, però en lloc d'utilitzar electricitat, modulen llum per identificar els bits.

FIGURA 4.1: Mòdem òptic Hydromea Luma X

Segons l'empresa que els fabrica, aquests mòdems són capaços d'intercanviar-se informació sempre que estiguin dintre del seu camp de visió de 120 graus, a una velocitat teòrica de fins a 10 Megabits per segon. A més a més, com que són pensats per a robots submarins, també poden aguantar la pressió de l'aigua fins a 6000 metres de profunditat.

4.2 CIRS i els submarins Girona

El CIRS (Centre d'Investigació Robòtica Submarina) és un centre d'investigació local associat amb la Universitat de Girona que experimenta amb tot el que sigui relacionat a la robòtica submarina, i és qui proporciona tot el material per a aquest projecte.

El centre disposa de diversos robots subaquàtics, com ara el Girona500 o el Girona1000, els quals són petits submarins autònoms (també es poden monitorar o teleoperar a través d'un cable), pensats per a la recerca marina. Aquests estan habilitats per dur diversos acoblaments, com ara càmeres, tota mena de sensors o fins i tot braços robòtics; o d'altres més atípics, com els mòdems òptics d'aquest mateix projecte, ja que funcionen sobre la plataforma ROS, que permet flexibilitat en el seu comportament.

La intenció d'aquest treball és desenvolupar un codi que permeti la comunicació entre un dels robots Girona sense cable mitjançant els mòdems Luma, fent ús de les instal·lacions del CIRS, que compten amb una piscina per fer aquesta classe de proves.

4.3 ROS

El Robot Operating System (ROS) és un framework potent i flexible de codi obert que proporciona una sèrie de llibreries i eines per ajudar als desenvolupadors a construir i controlar sistemes robòtics complexos. És la base sobre la qual treballen actualment la majoria dels projectes de recerca robòtica, inclosos els robots submarins del CIRS.

ROS aporta una arquitectura flexible i modular que permet escriure codi per a robots en múltiples llenguatges de programació, com ara Python, C++ o Java, a més a més

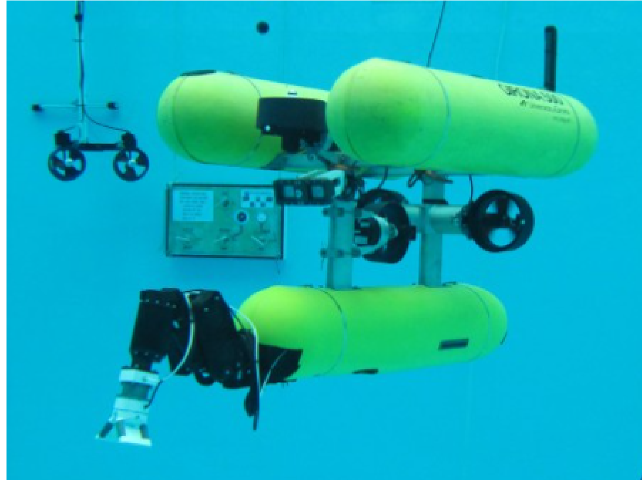

FIGURA 4.2: El robot Girona 500 submergit

d'una manera estandarditzada perquè diferents tipus de components de software es puguin comunicar, a través de sistemes publicador-subscriptor i servei-client.

El funcionament general dels sistemes que usen ROS consisteix en diversos fils d'execució anomenats "nodes", els quals s'encarreguen de fer una tasca específica, controlats per un programa principal anomenat *roscore*, el qual s'ocupa de comunicar tots aquests nodes entre si. Per tal de comunicar-se, aquests nodes, en lloc d'enviar-se informació directament, normalment fan servir un mètode de publicador-subscriptor, on un node pot penjar informació en un canal específic sota un cert nom, anomenats *topics*, els quals manega el *roscore*, i altres nodes poden accedir a aquests canals per consultar les dades proporcionades.

En un exemple pràctic, un node podria encarregar-se de controlar un motor d'un robot, mentre que un altre donar direccions al robot. En aquest cas, el node controlador podria ser un node publicador que pengés ordres de potència en un *topic* que es digués "/velocitat_motor", mentre que el node que controla el motor podria ser un subscriptor que estigués pendent d'aquest *topic*, executant totes les ordres que veiés d'aquest canal. Si un altre node volgués controlar el motor, seria capaç publicant comandes en el mateix *topic*. O si un altre node volgués consultar aquestes comandes, també podria escoltar-les subscrivint-se al canal amb nom "/velocitat_motor".

ROS també suporta un protocol de comunicació entre nodes de tipus servei-client, on, en lloc de parlar per canals públics on qualsevol node pot interactuar, un node escolta peticions (servei) d'altres nodes (clients), i aquest respon a cadascun individualment segons el contingut de la seva petició.

4.4 Protocols sobre IP

L'intercanvi de dades entre màquines no es redueix a res més que l'enviament d'uns i zeros per un canal de comunicació que les connecten. Aquests bits, però, no poden viatjar sols sense cap mena de control, ja que seria impossible identificar quan comença un tall d'informació, quan acaba, o saber a qui va dirigida. Per aquest fet, per tal de parlar-se entre si, les màquines usen protocols de xarxa, els quals empaqueten les dades de manera que es poden distingir com a blocs separats, anomenats

paquets, que, a més de les dades en si, contenen informació rellevant com la direcció on es pretén enviar, les seves dimensions, bits de control d'errors, etc.

El protocol de baix nivell més conegut i utilitzat és l'Internet Protocol (IP), on s'identifiquen els destinataris dels missatges una adreça de xarxa. Sobre aquest protocol es construeixen la majoria dels protocols emprats en les xarxes actuals, inclosos els que es tractaran en aquest treball, com són TCP, UDP, i RTSP.

Capítol 5

Requisits del sistema

5.1 Requisits funcionals

El sistema desenvolupat ha de permetre:

- Comunicar nodes de ROS a través dels módems optics de manera eficient, permetent tant el sistema de *topics* com el de serveis.
- Poder ser aplicat de manera genèrica a qualsevol projecte.
- Poder ser personalitzat per l'usuari, escollint quines funcionalitats usar.

5.2 Requisits no funcionals

Els requisits no funcionals variaran molt segons com s'utilitzi el software desenvolupat. En el cas d'aquest projecte, es provarà sobre una connexió a través dels mòdems òptics aquàtics LumaX, així que els seus requisits seran els següents:

- Dues màquines amb ROS Noètic instal·lat i interfície de xarxa Ethernet.
- Un contenidor d'aigua prou gran com per simular l'entorn de treball típic dels mòdems òptics.
- Mòdems òptics Hydromea Luma X, i els cables que els permeten ser connectats a ports Ethernet convencionals.

Capítol 6

Estudis i Descisions

6.1 Python i C++

Els programes que executa ROS poden funcionar tant en el llenguatge de programació Python, com en C++. Per aquest projecte es va escollir treballar únicament en Python, gràcies a la seva comoditat d'ús i al fet que, gràcies a estudiar la branca de robòtica, era el que es tenia més per mà.

Tot i això, el programa que s'acaba desenvolupant conté codi en C++, prestat del paquet ROS_RTSP [3].

6.2 OpenCV

OpenCV (Open Source Computer Vision Library) és una extensa llibreria de codi obert centrada en la visió per computador i el *machine learning*. Conté nombroses funcions i algorismes que poden ser utilitzats per a tasques de processament d'imatge, visió per computador, i aprenentatge d'intel·ligencies artificials.

S'ha emprat aquesta llibreria en el codi d'aquest projecte per processar totes les dades d'imatge i vídeo, tant per ser enviades com per ser capturades.

Cal mencionar que, pel testatge de vídeo s'ha fet servir el paquet de codi obert *video_stream_opencv*[4] per capturar el vídeo i transformar-lo a *topic* de ROS.

6.3 ROS_RTSP

Per implementar la comunicació de vídeo de manera eficient, explicat en profunditat en les seccions 8 i 9, s'ha integrat un servidor de vídeo RTSP en el codi del projecte.

El codi usat prové del paquet de codi obert ROS_RTSP, creat pel desenvolupador CircusMonkey[3].

Aquest paquet depèn de la llibreria Gstreamer per funcionar, la qual és necessària instal·lar per l'ús del programa desenvolupat en aquest projecte.

Gstreamer proporciona un marc de treball de codi obert centrat en les transaccions multimèdia, el qual s'enfoca en l'ús de *pipelines* per crear aplicacions multimèdia,

com la que s'acaba utilitzant per comunicar vídeo d'una màquina a una altra. Gstreamer també disposa de mètodes de compressió d'imatge i vídeo, factors claus en el funcionament correcte de l'enviament de vídeo d'aquest projecte.

En concret, el mètode de compressió emprat finalment en el codi del projecte ha estat el H.264. El funcionament d'aquesta mena de mètodes de compressió s'explica a la secció 8.5.2.

6.4 Protocols sobre IP

En aquest treball es tractarà l'ús dels següents protocols de xarxa:

6.4.1 Protocol TCP

El Transmission Control Protocol (TCP) és un dels protocols més usats a nivell global, ja que ofereix una manera de compartir dades entre màquines de manera fiable i ordenada, fent servir un canal de comunicació enfocat a la connexió entre els dispositius.

Una connexió TCP segueix els següents passos:

1. La màquina que vol començar la comunicació envia un paquet "SYN"(que vol dir *syncronize*) a la màquina que actua com a servidor, la qual espera peticions TCP.

2. El servidor rep el missatge i respon a l'origen amb un segment "SYN-ACK"(*synchronize-acknowledge*), indicant que accepta la connexió.

3. El client rep el paquet del servidor i respon amb el seu propi paquet "ACK"(*acknowledge*), per indicar al servidor que ha rebut el missatge correctament i el flux de missatges pot començar.

4. Llavors, qualsevol de les màquines involucrades pot enviar dades a l'altre, seccionant la informació en paquets TCP els quals contenen, a més a més de la informació principal, un nombre de seqüència per tal que el receptor pugui reconstruir el missatge original en ordre, el port destí, per poder diferenciar missatges de la mateixa font de diferents processos, i altres, com la mida de la informació.

5. En rebre un paquet, el dispositiu llença un segment "ACK"per confirmar la recepció existosa. Si no es rep aquest paquet dins d'una finestra de temps determinada, la màquina remitent torna a enviar el paquet fins que rep la confirmació.

6. Si s'ha acabat la seqüència de la comunicació i es vol alliberar el canal, una de les màquines envia un segment "FIN", al qual l'altre respon amb un "ACK". Llavors les dues màquines terminen la connexió enviant segments "FIN"fins que totes dues han rebut un "ACK"de l'altra part.

La popularitat d'aquest protocol es deu al fet que aporta una comunicació molt estable on, si es perd algun paquet, es pot recuperar fàcilment, ja que el remitent el tornarà a enviar si no rep el missatge "ACK", i es podrà utilitzar igualment en el

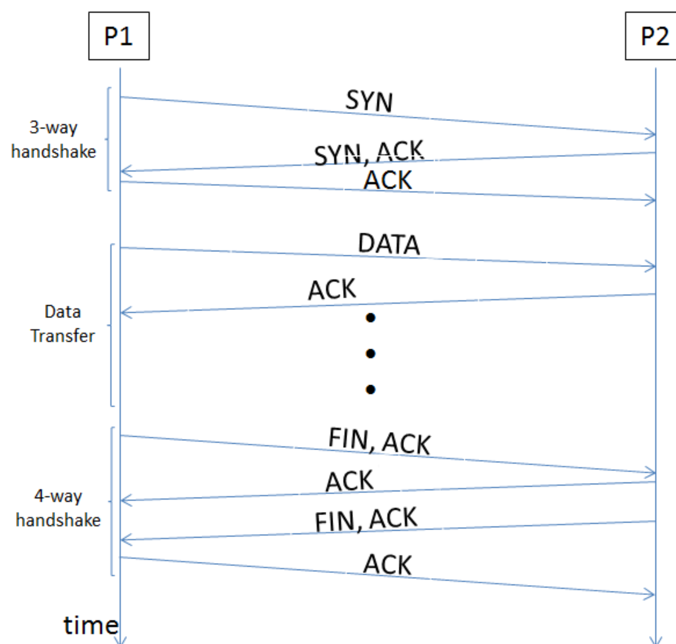

FIGURA 6.1: Diagrama d'una interacció TCP

mateix ordre, ja que tots els paquets van numerats. Aquest fet és molt útil en la majoria de xarxes, on només es perden un de cada molts paquets, i fins i tot en xarxes poc fiables, perquè asseguren la recepció. L'únic inconvenient és que, en xarxes amb molts d'erros, o per fluxos de dades on no és imprescindible rebre tota la informació, la pèrdua de paquets pot implicar un augment de la latència dramàtic, ja que s'intentarà reenviar tots els paquets que no s'han rebut abans d'enviar-ne de nous. Si se'n reben menys dels que es reenvien, es podria arribar a saturar per complet la línia.

6.4.2 Protocol UDP

L'User Datagram Protocol (UDP) és un dels protocols més senzills que hi ha sobre IP. En lloc d'enfocar-se a establir una connexió amb els altres dispositius, com fa TCP, aquest protocol envia el seu missatge directament a l'adreça IP desitjada, sense esperar cap mena de resposta.

UDP no assegura que el receptor rebrà els paquets enviats, però, a canvi, proporciona una velocitat incomparable, ja que no s'han d'enviar cap altra mena de paquets abans de transmetre la informació principal i, a més, la capçalera dels seus missatges, on es troba tota la informació pertinent a l'enviament del paquet, és més petita que en els paquets TCP, fet que alleugera l'ús d'amplada de banda i el temps de processat.

Per tant, aquest protocol s'usa sobretot en aplicacions on la latència baixa i la velocitat es prioritzen per sobre de la correcta recepció de tots els paquets.

6.4.3 Protocol RTSP

Per aquest projecte també es farà ús del protocol RTSP (Real Time Streaming Protocol), un protocol que col·labora amb altres, com ara TCP i UDP i, sobretot RTP (Real

Time Protocol), per transmetre fluxos continus de dades en temps real, com poden ser els vídeos en directe.

En una connexió RTSP, un dispositiu actua com a servidor, el qual envia les dades segons les comandes que rep de la màquina client. Una conversa RTSP segueix la següent estructura:

1. El client envia una comanda "SETUP"per RTSP al servidor. Aquesta comanda especifica quins protocols es vol utilitzar (normalment RTP sobre UDP o RTP sobre TCP), els ports del client on es vol rebre el flux de dades.

2. Establert per quin canal es rebran les dades, el client envia una comanda "PLAY"per ordenar al servidor que inicií la transmissió de les dades del recurs especificat en aquest mateix paquet.

3. El servidor envia les dades demanades sobre el protocol preestablert, fins que rep, o bé una comanda "PAUSE", la qual indica parar l'enviament, o una comanda "TEARDOWN", que indica el fi de la comunicació amb el client.

El servidor usa normalment RTCP (Real Time Control Protocol) per enviar el contingut demanat, a més a més del protocol d'enviament especificat. Aquest protocol li permet recol·lectar estadístiques de la connexió, com ara la pèrdua de paquets, la latència, la variació en el temps entre l'enviament i la recepció, d'entre altres. D'aquesta manera, el servidor és capaç de controlar el seu enviament de dades per acomodar-se a la connexió amb el client i mantenir una latència baixa, característica d'una transmissió de temps real.

Capítol 7

Anàlisi i disseny del sistema

7.1 Comunicació estàndard de ROS

Per comunicar-se entre si, una parella operador-robot típica es parla a través d'un sistema de publicador-subscriptor, amb *topics* que proporciona la plataforma ROS. Els *topics*, o temes, són com una mena de canals de comunicació etiquetats on els nodes publicadors poden penjar informació. Els publicadors són funcions que s'encarreguen de penjar informació en els temes, mentre que els subscriptors s'encarreguen de llegir-ne les dades que s'hi publiquen. Essencialment, les converses d'aquesta mena consisteixen en que un publicador penja unes dades en el seu tema, els subscriptors d'aquest tema són despertats, llegeixen què s'ha publicat i llavors actuen segons què han rebut.

En la majoria dels casos, en la comunicació entre l'operador i el robot existeixen múltiples publicadors i subscriptors de ROS. Per exemple, per llegir una imatge generada per la càmera d'un robot, se segueix el següent procés:

1. El robot crea un publicador encarregat de penjar les dades de la imatge.
2. L'operador genera un subscriptor que se subscriu al tema de la imatge.
3. La càmera genera una imatge.
4. El publicador del robot llegeix aquesta imatge i la publica al seu tema.
5. El subscriptor rep l'últim missatge del tema i processa la informació (en el cas de la imatge, la mostra o la guarda en un fitxer).

Aquest protocol és de gran utilitat en sistemes robòtics, on a diversos components els interessa conèixer informació d'altres, però, per la part d'enviament de dades, els protocols que s'usen impliquen un major volum de paquets enviats, en comparació a simplement enviar les dades directament al receptor interessat.

Un altre mètode que tenen els nodes de ROS per comunicar-se és el de client-servidor. En lloc d'utilitzar canals públics, on qualsevol subscriptor pot rebre'n la informació, en aquesta mena de comunicacions, els nodes que fan la funció de client realitzen una crida al node servidor d'interès. Aquest, en rebre la crida, responen amb l'acció sol·licitada, ja sigui retornar informació al node client, o fer una acció en específic.

En un exemple pràctic, un servei d'un robot podria encarregar-se de retornar un diagnòstic d'aquest. El procés seria el següent:

1. El robot engega un servei específic que retorna diagnòstics. Aquest servei es queda a l'espera de ser cridat.

2. L'operador genera una crida a través d'un node client, sol·licitant l'acció del servei.

3. El servei rep la comanda i executa la funció per tractar-la. En aquest cas, genera el diagnòstic.

4. Un cop generat, el servei respon al client amb el missatge que ha creat.

Els serveis estan pensats per ser usats de manera puntual, on l'execució correcta és crucial, mentre que els *topics* són canals on es penja informació més recurrent.

7.2 Enviament de paquets en ROS

Per comprovar el consum d'amplada de banda que pot suposar fer servir el sistema de publicadors i subscriptors de ROS, es va començar a experimentar amb dos segments de codi senzills, executats en màquines virtuals, per simular una comunicació interna de ROS.

Es van crear dos codis bàsics de comunicació, un que fes la funció de publicador, i l'altre de subscriptor:

- Node publicador:

  - El codi del node publicador comença iniciant el mateix node, amb el nom de "talker", perquè ROS l'identifiqui.

  - Llavors, es crea el mateix publicador amb la comanda *rospy.Publisher()*. Aquest publicador és enllaçat amb el *topic* çhatter", on es pengen missatges de tipus String. Tots els missatges que pengi aquest publicador seràn formatats com a fils de caràcters i aniràn al *topic* çhatter". Qualsevol node podrà consultar aquest *topic* i rebre la informació que comparteixi el publicador.

  - Per últim, s'entra en un bucle, el qual mana al publicador publicar informació cada 10 segons, fins que s'apagui el *roscore*. Se li comanda penjar poques dades, en aquest cas el caràcter "a", que ocupa un byte, ja que interessa observar el consum d'amplada de banda addicional a la càrrega principal dels paquets enviats. D'aquesta manera, cada 10 segons al *topic* çhatter"apareixerà un String que conté el caràcter "a".

- Node subscriptor:

  - El node subscriptor s'inicia de la mateixa manera que l'anterior, posant en marxa el node. Aquest sota el nom de l·listener".

  - Seguidament, es crea el subscriptor amb la comanda *rospy.Subscriber()*, el qual es subscriu al canal "/chatter", on el node "talker"parla. Quan el

```
7  def talker():
8      pub = rospy.Publisher('chatter', String, queue_size=10)
9      rospy.init_node('talker')
10
11     while not rospy.is_shutdown():
12         str = "a"
13         rospy.loginfo(str)
14         pub.publish(str)
15         time.sleep(10)
```

FIGURA 7.1: Codi que executa el node publicador simple

subscriptor llegeixi un missatge d'aquest *topic*, processarà la informació cridant la funció *callback()*. Aquesta funció simplement mostra per pantalla el que ha rebut.

– Per acabar, s'executa la comanda *rospy.spin()*, que mante el programa en execució fins que s'acabi l'execució de ROS. Llençar aquesta funció és necessari per mantenir el subscriptor engegat.

```
5  def callback(data):
6      rospy.loginfo("HE REBUT: \n %s", data)
7
8  def listener():
9      rospy.init_node('listener', anonymous=True)
10
11     rospy.Subscriber("/chatter", String, callback)
12
13     rospy.spin()
```

FIGURA 7.2: Codi que executa el node subscriptor simple

Mitjançant la eina Wireshark, que permet capturar paquets de xarxa de tota mena, inclosos els interns en la mateixa màquina, podem observar el comportament de les comunicacions de ROS.

Per començar, iniciant els nodes descrits, es pot veure com ja s'intercanviaven prop de 5 kilobytes per cadascun, ja que el codi que els genera parla amb el *roscore* per fer-ho. No és un volum significatiu, perquè només es reproduiria una vegada per creació de node, però per a l'aplicació d'un projecte on enviar pocs paquets per un enllaç fos essencial, també seria informació que hauria de passar per l'enllaç que podria ser estalviada fent servir *roscores* separats.

Seguidament, si esperem que el publicador envií un dels seus missatges, on la càrrega útil és d'1 byte, es capturen un total de 16 paquets, que equivalen a una conversa de prop de 3 kilobytes.

Sembla una quantitat excessiva, i, de fet, ho és. Tota aquesta conversa es genera pel fet que, tant el subscriptor com el publicador, fan ús de la funció *rospy.loginfo()* per notificar el seu comportament. Aquesta funció, tot i que molt útil per mediar entre el *roscore* i l'usuari, implica un augment de l'ús d'amplada de banda, ja que cada impressió per pantalla suposa més d'un kilobyte d'informació intercanviada entre el node i el *roscore*, sense tenir en compte el mateix missatge imprès.

Si es substitueixen aquestes crides per impressions normals de Python amb la funció

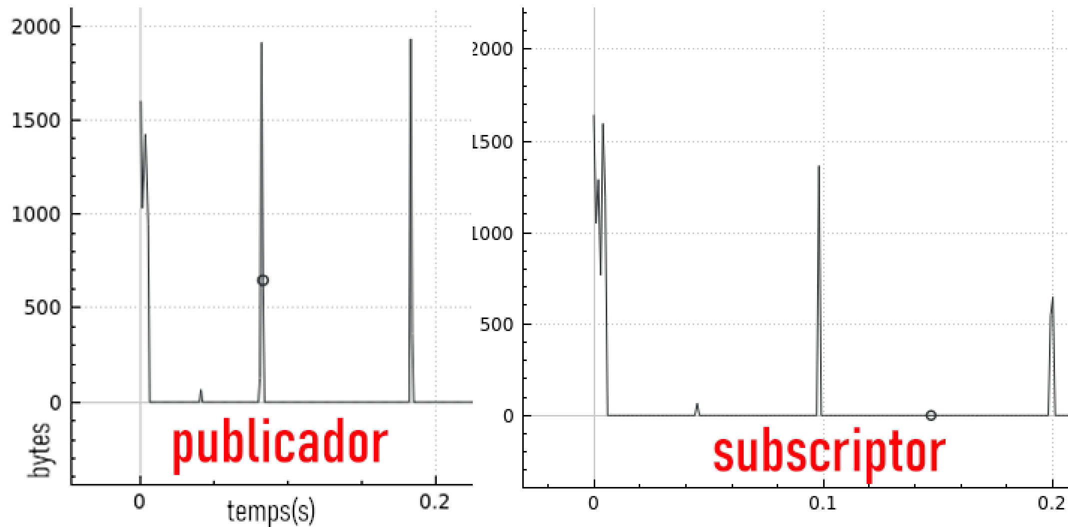

FIGURA 7.3: Gràfic de l'ample de banda usat en iniciar els nodes subscriptor i publicador simples

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	8.8.8.4	8.8.8.4	TCP	209	49530 → 11311 [PSH, ACK] Seq=1 Ack=1 Win=512 Len=143 TSval=21…
2	0.000020696	8.8.8.4	8.8.8.4	HTTP/X…	404	POST /RPC2 HTTP/1.1
3	0.000144403	8.8.8.4	8.8.8.4	TCP	66	11311 → 49530 [ACK] Seq=1 Ack=482 Win=512 Len=0 TSval=2145976…
4	0.000718450	8.8.8.4	8.8.8.4	HTTP/X…	495	HTTP/1.1 200 OK
5	0.000722849	8.8.8.4	8.8.8.4	TCP	66	49530 → 11311 [ACK] Seq=482 Ack=430 Win=509 Len=0 TSval=21459…
6	0.001112266	8.8.8.4	8.8.8.4	TCP	184	39859 → 37278 [PSH, ACK] Seq=1 Ack=1 Win=512 Len=118 TSval=21…
7	0.001119434	8.8.8.4	8.8.8.4	TCP	66	37278 → 39859 [ACK] Seq=1 Ack=119 Win=507 Len=0 TSval=2145976…
8	0.001250979	8.8.8.4	8.8.8.4	TCP	75	39859 → 44962 [PSH, ACK] Seq=1 Ack=1 Win=512 Len=9 TSval=2145…
9	0.001254217	8.8.8.4	8.8.8.4	TCP	66	44962 → 39859 [ACK] Seq=1 Ack=10 Win=511 Len=0 TSval=21459763…
10	0.001766154	8.8.8.4	8.8.8.4	TCP	209	47706 → 11311 [PSH, ACK] Seq=1 Ack=1 Win=512 Len=143 TSval=21…
11	0.001779450	8.8.8.4	8.8.8.4	HTTP/X…	393	POST /RPC2 HTTP/1.1
12	0.002042516	8.8.8.4	8.8.8.4	TCP	66	11311 → 47706 [ACK] Seq=1 Ack=471 Win=512 Len=0 TSval=2145976…
13	0.002382293	8.8.8.4	8.8.8.4	HTTP/X…	495	HTTP/1.1 200 OK
14	0.002385342	8.8.8.4	8.8.8.4	TCP	66	47706 → 11311 [ACK] Seq=471 Ack=430 Win=509 Len=0 TSval=21459…
15	0.002968558	8.8.8.4	8.8.8.4	TCP	196	43941 → 44310 [PSH, ACK] Seq=1 Ack=1 Win=512 Len=130 TSval=21…
16	0.002972053	8.8.8.4	8.8.8.4	TCP	66	44310 → 43941 [ACK] Seq=1 Ack=131 Win=507 Len=0 TSval=2145976…

FIGURA 7.4: Paquets capturats en publicar i llegir un sol missatge i notificant-ho amb la comanada *rospy.loginfo()*

print(), i es torna a capturar els paquets enviats quan el publicador penja un caràcter en el seu *topic*, es veu com la conversa queda reduïda a un enviament i una resposta de TCP.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	8.8.8.4	8.8.8.4	TCP	75	34287 → 39170 [PSH, ACK] Seq=1 Ack=1
2	0.000007755	8.8.8.4	8.8.8.4	TCP	66	39170 → 34287 [ACK] Seq=1 Ack=10 Win:

FIGURA 7.5: Paquets capturats en publicar i llegir un sol missatge i notificant-ho amb la comanada *print()*

D'aquest comportament se'n pot deduir que, quan es genera el subscriptor, aquest estableix una connexió TCP amb el node publicador, i, quan el publicador penja dades a un *topic*, el que fa és enviar pels canals que té oberts la informació.

Si, en lloc d'un subscriptor, se'n fan servir dos, el que es captura en publicar un missatge són exactament dues converses de TCP, confirmant aquest comportament.

Per tant, el *roscore* funciona com a mediador entre els nodes mitjançant els *topics*, negociant les connexions TCP entre si quan s'inicien els nodes, vist en la conversa de la figura 7.3, i llavors aquests nodes poden conversar sense problemes un cop creats els seus canals personals.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	8.8.8.4	8.8.8.4	TCP	75	34287 → 39170 [PSH, ACK] Seq=1 Ack=1 Win=512 Len=
2	0.000008024	8.8.8.4	8.8.8.4	TCP	66	39170 → 34287 [ACK] Seq=1 Ack=10 Win=511 Len=0 TS
3	0.000040762	8.8.8.4	8.8.8.4	TCP	75	34287 → 51144 [PSH, ACK] Seq=1 Ack=1 Win=512 Len=
4	0.000043496	8.8.8.4	8.8.8.4	TCP	66	51144 → 34287 [ACK] Seq=1 Ack=10 Win=511 Len=0 TS

FIGURA 7.6: Paquets capturats en publicar un sol missatge i ser capturat per dos subscriptors

Per aquest projecte, l'ús de TCP no és ideal, a més a més que interaccions com les que s'han vist a la figura 7.4 no ajuden. A la següent secció s'entra en detall del que es va fer coneixent el comportament de ROS.

7.3 Aplicació particular

Per a l'aplicació d'aquest projecte en concret, és força irrellevant tot l'enviament de dades que hi hagi internament dins de les màquines. El que interessa és enviar la menor quantitat de dades com sigui possible pels mòdems, i evitar retransmissions innecessàries.

Per tant, el que es va buscar fer primer va ser eliminar la comunicació del ROS entre les màquines unides pels Luma, substituint-la per un protocol de comunicació més senzill, que prioritzés la latència baixa davant de la qualitat de les dades. Per a aquesta tasca no és necessari un enviament totalment fiable de dades, si s'escullen els protocols adequats, ja que les dades que s'intenten transmetre, la majoria són d'un flux continu, com ara vídeo, comandes de moviment o odometria, així que si es perd algun paquet pel camí, o alguna de les dades és errònia a causa d'una pertorbació en el transport, la qualitat no es veurà tan afectada, ja que la informació de seguida és reescrita. El que és important és tenir una latència baixa, per poder seguir tots els moviments i accions en temps real.

Com que ROS no permet triar com es parlen els subscriptors i publicadors amb els *topics*, el que es va optar va ser d'aïllar l'ús de ROS dintre de les màquines. El que es va idear va ser fer un canal de comunicació personalitzat entre els dos *roscores*, replicant el comportament d'un sol *roscore*, però fent servir un protocol d'enviament molt més senzill com pot ser UDP, escollint especificament quines dades es volen transmetre entre màquines. Així, processos interns de les màquines que usen ROS podrien seguir funcionant sense problemes, però sense usar amplada de banda dels mòdems òptics si el *roscore* es troba a la màquina de l'altra banda de la connexió.

D'aquesta manera va començar el desenvolupament del codi principal d'aquest treball, el paquet ROS2ROS.

7.4 Concepte rere el ROS2ROS

Nativament, una execució de ROS supervisa a tots els seus components, i aquests només parlen amb un sol *roscore*. Si una màquina externa a la qual els nodes tenen connexió també engega un *roscore*, amb els seus *topics* pertinents, els nodes de la primera màquina no podràn veure els *topics* de la segona, i viceversa. Un node només pot comunicar-se amb un *roscore* a la vegada, que és qui supervisa tots els *topics* locals i els seus subscriptors i publicadors.

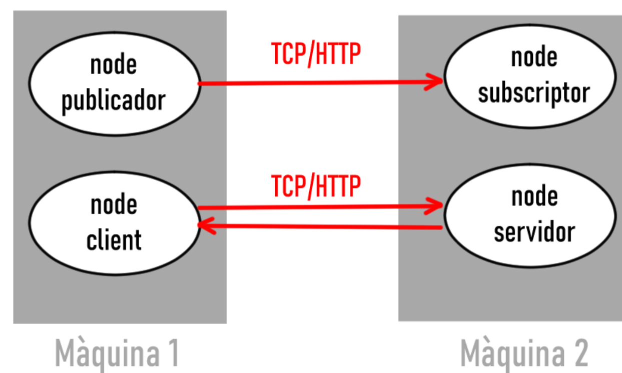

FIGURA 7.7: Diagrama de la comunicació entre nodes de màquines separades executats sota un mateix *roscore*

Així doncs, com es pot fer que dues execucions separades de ROS es parlin? La clau és en els subscriptors i publicadors, els quals, tot i no tenir una connexió directa al *roscore* veí, poden parlar amb la màquina que l'executa a través d'altres canals.

La idea principal darrere el paquet que s'ha desenvolupat per a aquest treball ha estat que, per cada *topic* que pot gestionar un *roscore*, i que interessa comunicar-lo a l'altre, crear una parella de subscriptor i publicador "fantasmes"que facin la feina de parlar entre les màquines. En essència, el que s'ha fet és, per una banda, generar un subscriptor a un *topic*, especificat prèviament per un arxiu de configuració, el qual s'encarrega de traduir tot el que es transmet per aquell *topic* a missatges que s'envien a l'altra màquina un cop rebuda informació. Per l'altra banda de la connexió, un publicador espera rebre aquests missatges per convertir-los al seu format original i publicar-los en un *topic* administrat ja per l'altre *roscore*.

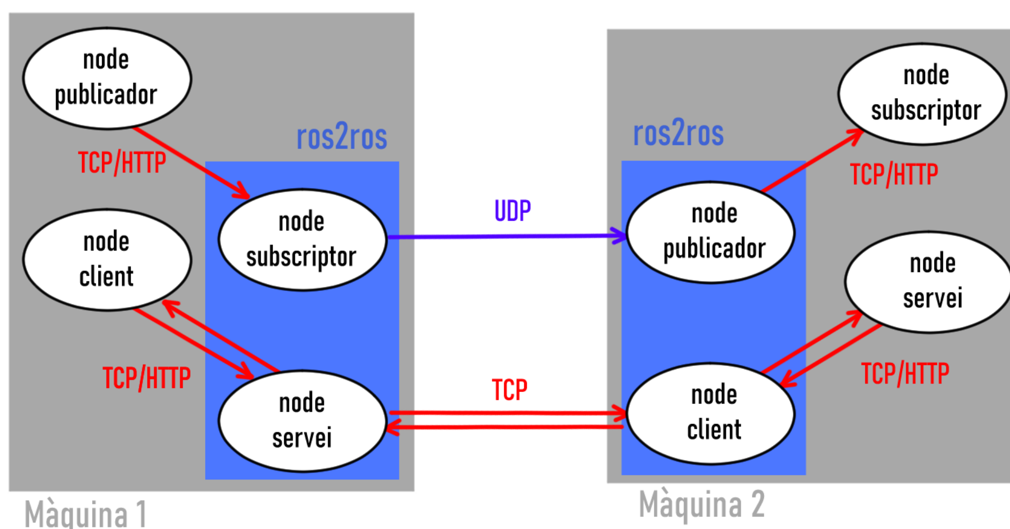

FIGURA 7.8: Diagrama del funcionament del ROS2ROS

D'aquesta manera, un publicador d'una màquina penja el seu missatge, és llegit pel subscriptor del ROS2ROS, i el missatge és enviat a l'altra màquina, on és retransmès pel publicador que rep la informació, de manera que el missatge queda penjat en el *topic* de les dues màquines, funcionant com un de sol.

Aquest principi també s'aplica als serveis, però, per la seva naturalesa, on prima el correcte enviament de les peticions, no la velocitat, la comunicació es fa per TCP en lloc d'UDP.

La implementació final i el seu funcionament en detall es pot veure a l'apartat 10.

Capítol 8

Implementació i Desenvolupament

8.1 Primers passos

L'aplicació ROS es basa en converses entre fils d'execució independents anomenats nodes. Per parlar-se, aquests nodes usen missatges sobre protocols HTTP i TCP. Aquests protocols proporcionen una comunicació segura i robusta, però són pensats per a una connexió relativament estable amb un amplada de banda convencional (>100MBits). Si no reben un paquet, o el reben amb errors que no es poden corregir, l'emissor el torna a enviar, una vegada i una altre fins que és rebut correctament (o fins a arribar a un límit de retransmissions). Aquest fet pot resultar útil per a xarxes que perden un de cada molts paquets, però, si hi ha pèrdues importants, els receptors poden acabar demanant tantes retransmissions de paquets que han perdut que la xarxa es satura de reenviaments, resultant en un augment de la latència exponencial.

Els mòdems òptics LUMA funcionen, tècnicament, de manera transparent a la xarxa. És a dir, són com un cable més de transport entre punt i punt de la xarxa. A la pràctica, però, donen una connexió molt menys sòlida, ja que depenen de molts més factors externs que un simple cable d'Etherent, com l'angle en què s'estan mirant, la intensitat de la llum que emeten, la distància a la qual estan, i, sobretot, la llum externa, la qual pot causar interferències.

Abans de començar a provar les seves capacitats, ja es preveia que la connexió dels mòdems causaria moltes pèrdues de paquets, cosa que pot dur molts problemes tenint en compte el funcionament de ROS. Així doncs, aquest treball va començar el seu recorregut investigant com lidiar amb la pèrdua de paquets entre les màquines quan s'utilitzessin els mòdems LUMA.

En una xarxa amb pèrdues, a més dades s'enviïn, més dades seran errònies o perdudes per culpa de la mala connexió. Per tant, el primer pas que es va fer en aquest projecte va ser el d'idear una manera de reduir la informació que s'intercanvien les màquines a través dels mòdems.

8.2 Primeres proves locals

8.2.1 Testatge de *topics*

Un cop implementada una versió inicial del paquet ROS2ROS, la qual permetia comunicar *topics* via UDP, el primer que es va realitzar va ser de nou la prova del

subscriptor i publicadors simples. Aquesta vegada, però, utilitzant màquines virtuals separades dins la mateixa xarxa, on cada node es troba a una màquina diferent. D'aquesta manera, es poden filtrar els paquets que no s'estiguin enviant entre nodes, obviant tots els altres missatges interns de ROS que es poden originar dintre dels dispositius.

No.	Time	Source	Destination	Protocol	Length	Info
5	6.491781230	8.8.8.4	8.8.8.5	UDP	60	59520 → 10004 Len=5

FIGURA 8.1: Paquets capturats en publicar un missatge d'1 byte i comunicar-lo entre màquines per UDP

En la figura anterior es pot apreciar com el fet d'usar el ROS2ROS alleugera la càrrega de la comunicació entre nodes. En la figura 7.5 s'ha vist que amb un sol *roscore*, aquesta conversa suposa un tràmit de dos missatges TCP, per un total de 141 bits enviats. Traient els bytes d'informació útil del missatge (que per com processa les publicacions ROS són 5 bytes), els 136 restants són bytes de capçalera i confirmació que s'utilitzen en el protocol de TCP per a cadascuna de les transaccions. Fent servir el paquet ROS2ROS, aquesta interacció es redueix a només 60 bytes, on 55 són part de la capçalera del missatge. Per tant, convertint la comunicació de ROS de TCP a UDP, enfocant-se només en la comunicació entre nodes, es pot reduir l'amplada de banda utilitzat per les dades de protocol, que no formen part de la informació principal, a gairebé un terç, a cost de no tenir la possibilitat de recuperar els paquets que es perdin pel camí. Pel caràcter de les dades que s'han de tramitar amb els mòdems òptics en l'aplicació d'aquest projecte, aquest és un preu assumible, i potser fins i tot beneficiós, ja que s'evita el desbordament que poden ocasionar les retransmissions del protocol TCP quan es perden molts paquets.

Es van dur a terme altres proves, per comprovar que tota mena de paquets es poden passar sense problemes. D'aquestes, es poden destacar les proves que es van realitzar fent servir un *topic* que transmetés imatge contínua. Fent servir un paquet d'Opencv que transforma el vídeo de la càmera de l'ordinador en *topics* de ROS, es pot veure com l'amplada de banda ocupat per transmetre imatge comprimida, amb resolució 640x480 a 30 imatges per segon, ronda els 0,6 Megabytes per segon tant si s'utilitza UDP com ROS amb TCP.

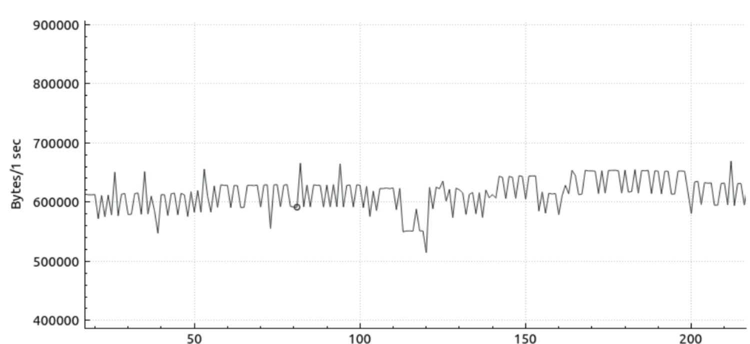

FIGURA 8.2: Ample de banda utilitzat enviant imatge comprimida d'una *webcam* per UDP

D'aquests resultats se'n van poder uns quants fets:

- Cal utilitzar un vídeo gravat amb anterioritat, ja que la compressió de les imatges pot afectar molt l'amplada de banda emprat. En la figura 8.3 es pot veure

com el *bitrate* cau en picat al segon 75. Això es deu al fet que, en aquell instant, es va tapar l'obertura de la càmera, resultant en una imatge pràcticament negra, molt més fàcil de comprimir.

- Cal fer servir un mètode de compressió exprés per vídeo si es vol reduir l'amplada de banda utilitzat, que tingui en compte la dimensió temporal. Enviar cada fotograma del vídeo com una imatge, encara que sigui una imatge de baixa resolució i comprimida, l'ample consumit és molt elevat.

- Tot i que no es pot acabar de determinar degut a l'ús d'una font de vídeo diferent per cada prova, l'*overhead* dels protocols sembla ser insignificant en una connexió sense pèrdues. Cal experimentar amb els mòdems òptics per comprovar si les retransmissions de paquets TCP poden suposar un problema.

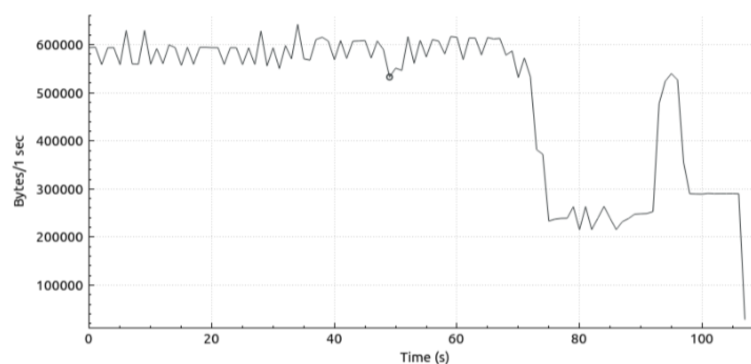

FIGURA 8.3: Canvi del pes de la retransmissió en tapar la càmera

8.2.2 Testatge de services

Per complir l'objectiu d'unificar dos *roscores* separats, no només cal que es parlin per *topic*. Una altra funcionalitat clau de ROS és la comunicació servei-client. Aquesta mena de comunicacions necessiten un node que estigui pendent a ser cridat, i d'un altre que el cridi.

Per provar la funcionalitat desenvolupada, es va implementar un servei molt senzill, el qual, en ser cridat, retorna un text confirmant la connexió. El seu codi es pot veure en la figura 8.4. Bàsicament, es crea un servei sota el nom de "/test_serviceï, quan una crida és dirigida al service sota aquest nom, la funció *trigger_response()* és executada.

En la secció 9.1.1 es mostra el correcte funcionament del ROS2ROS cridant a aquest servei des d'una màquina amb un *roscore* diferent. Es va decidir mantenir la comunicació per TCP perquè les crides als serveis solen ser puntuals i és important la bona recepció del missatge.

8.3 Instal·lació LUMA

Per fer les proves necessàries per desenvolupar el codi i observar el comportament dels mòdems òptics, tot i que l'ús real dels mòdems seria acoblats a un robot submarí, no era viable muntar els Luma a un robot i fer cada experiment amb aquest dintre de la piscina, ja que altres projectes estaven sent desenvolupats alhora en les instal·lacions del CIRS i necessitaven la piscina, a part que no era gens pràctic haver

```python
import rospy
from std_srvs.srv import Trigger, TriggerResponse

def trigger_response(request):
    return TriggerResponse(
        success=True,
        message="El service funciona!"
    )

rospy.init_node('sos_service')
my_service = rospy.Service('/test_service', Trigger, trigger_response)
rospy.spin()
```

FIGURA 8.4: Codi del servei simple

de posar el submarí sota d'aigua per cada petita prova de connexió que es volgués fer. Per això, es va recrear un entorn similar al de treball dels mòdems, arraconat a un costat del tanc d'aigua per no impedir-ne l'ús.

Per tal de simular aquest entorn de treball, es van submergir els mòdems, fixats a un maó per evitar que es moguessin, un a cada costat de la piscina, col·locats mirant-se a, aproximadament, uns 6 metres de distància. Aquesta distància és relativament curta per a l'aplicació dels mòdems, però es volia primer assegurar que la connexió fos estable entre ells abans de forçar les seves capacitats.

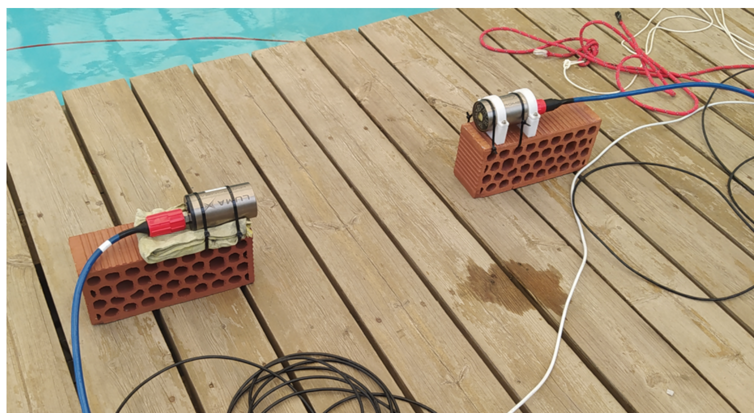

FIGURA 8.5: Mòdems òtpics Luma abans de ser submergits

Com que el programari dels robots es basa en ROS, no és necessari usar específicament un dels submarins per fer les proves. Utilitzar un ordinador amb ROS instal·lat és suficient per simular la funció del robot.

Per operar els Luma es van fer servir dues màquines Linux, un per cada mòdem, les quals tenien dues interfícies de xarxa. Una capaç de connectar-se a Internet, per tal de descarregar tot el que fos necessari i accedir a les màquines de forma remota, i una altra aïllada de totes altres connexions excepte les del mòdem. Connectats d'aquesta manera, si es vol comunicar els ordinadors a través dels Luma, es pot fer apuntant a la xarxa privada composta per les interfícies privades de les màquines i els mateixos mòdems, la qual està totalment aïllada de la resta de xarxes.

Els mòdems òtpics Luma X, tot i ser dissenyats per corregir interferències de llum,

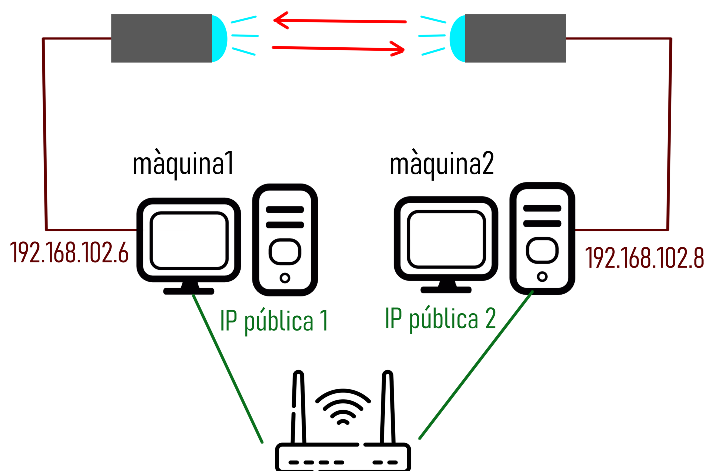

FIGURA 8.6: Diagrama de la connexió en l'entorn de proves

són pensats per ser usats a profunditats on la llum natural no arriba. Per evitar interferències de la llum exterior, es van aplicar dues mesures. Per a les primeres proves es va desplegar parcialment una cortina que està instal·lada a la piscina del CIRS, donant ombra a la secció de la piscina on es trobaven els mòdems.

FIGURA 8.7: Cortina de la piscina del CIRS tapant la zona de treball

Més endavant, després d'experimentar problemes de visibilitat entre els Luma, es va optar per, a més a més de la cortina, posar els mòdems dintre d'un cilindre que acabés d'enfosquir el seu camp de visió.

Per agilitzar el procés, com que els terminals fets servir consten d'una interfície amb accés a l'Internet, vist en el diagrama 8.6, es va instal·lar el programa AnyDesk en ambdues màquines, per poder accedir a aquestes remotament i realitzar totes les proves necessàries de forma telemàtica.

FIGURA 8.8: Cilindre de plàstic usat per tapar els mòdems òptics

8.4 Proves de banda d'amplada LUMA

Un cop instal·lat tot l'entorn del treball pràctic, es van realitzar múltiples proves d'amplada de banda. Abans, però, es va comprovar que les màquines es poguessin fer *ping* entre les elles, per assegurar que hi havia connexió. L'eina *ping* és una eina molt bàsica en el món de les xarxes. Quan un ordinador fa *ping* a una adreça IP, envia un petit paquet cada segon, el qual, si hi ha connexió i la màquina receptora té els pings habilitats, aquesta última respon amb una confirmació per cada paquet.

Havent establert la connexió, es va passar a examinar la qualitat de la xarxa dels mòdems mitjançant diferents mètodes, com ara l'ús l'eina Iperf, que permet posar a prova una connexió amb diferents mètodes predefinits, com fent proves manuals, o compartint missatges de text, d'imatge, i finalment, de vídeo. Els resultats d'aquestes proves són documentats a la secció 9.

Després de solucionar diversos problemes que van anar sorgint durant les proves, els quals impedien la comunicació entre els mòdems i van suposar moltes sessions de resolució, com ara una instal·lació defectuosa dels cables que connectaven les màquines amb els Luma, o l'excés de llum provinent de fora de la piscina, els fets que es poden destacar són els següents:

- Es va observar que tots els missatges enviats de més, un cop l'amplada de banda dels Luma està totalment utilitzat, es comencen a descartar paquets. Això es dèu a que internament els mòdems tenen uns buffers limitats. Es va observar que Linux, així com altres sistemes operatius moderns, assumeixen que un cable de xarxa ofereix, com a mínim, 100 Mbits d'amplada de banda. Això, contraposat als 10 Mbits teòrics dels Luma, causa que ràpidament el superin els limits dels buffers dels Luma. Per solucionar això, és necessari limitar manualment l'amplada de banda màxim que pot fer servir la màquina per parlar amb els Luma, forçant que sigui l'ordinador qui fa buffering i no el mòdem.

- Els mòdems òptics Luma, tot i funcionar de manera transparent a la xarxa, ofereixen una interfície web local que permet accedir a la configuració dels

mateixos mòdems. Es poden modificar una quantitat considerable de paràmetres, d'entre els quals, els que van tenir un efecte més apreciable van ser la freqüència d'enviament, el nombre de LEDs emissors, el nombre de receptors de llum, i el guany de la llum rebuda. Aquest últim paràmetre determina el factor pel qual es multiplica la quantitat de llum rebuda. Es pot modificar manualment per filtrar les possibles pertorbacions de la llum externa, o deixar que el programari del mòdem s'encarregui de modificar-lo segons el què rebin els seus sensors.

- Es va realitzar una millora del codi per poder triar de manera personalitzada quins *topics* enviar per UDP, quins per TCP, i quins per RTSP (aquest últim s'explica en detall en l'apartat 8.5. El codi, originalment només permetia compartir la informació dels *topics* per UDP, i les peticions dels serveis per TCP.

8.5 Transmissió de vídeo

8.5.1 Enviament d'imatge contínua

En comprovar que els mòdems són capaços d'enviar-se dades petites, com enviar-se text o fer *ping* entre les màquines, es va passar a intentar transmetre volums més considerables d'informació, principalment enfocant-se en el vídeo en directe, per poder captar el què veu una càmera del robot.

Com es va veure durant les proves locals, és necessari utilitzar un vídeo grabat amb anterioritat, per evitar que la compressió afecti l'amplada de banda usat si la informació de la imatge canvia, fent ambigús els resultats. Partint d'una gravació d'una escena d'una carretera amb trànsit en moviment, es va fer ús el paquet de vídeo d'OpenCV per convertir-lo a un *topic* de ROS, i es va intentar retransmetre entre les màquines del CIRS amb el paquet ROS2ROS. Es va triar un vídeo d'una carretera congestionada en lloc d'una gravació d'una càmera sota el mar, que en el context d'aquesta aplicació tindria més sentit, per tal de captar fàcilment si el vídeo concorda amb el que es rep a la segona màquina, i, de pas, observar millor les limitacions dels mòdems, ja que si poden transportar vídeo amb molta diferència, també són capaços de transmetre vídeo amb menys moviment, més fàcils de comprimr.

De la mateixa manera que ja s'ha vist a l'apartat anterior, enviar imatge continua resulta molt costós, encara que sigui comprimida. En provar-ho, els mòdems van ser capaços de compartir-se els primers fotogrames del directe, però ràpidament es van veure sobrepassats per tota la informació a enviar i, tot i treballar en UDP, que no contempla retransmissions de paquets perduts, el directe es va quedar molt enrere del vídeo en temps real emès a la màquina original, mostrant molt pocs fotogrames per segon.

8.5.2 Compressió temporal

Per tenir alguna possibilitat d'aconseguir transmetre vídeo, era imperatiu alleugerar la càrrega d'aquest. Fins aquest instant, tota compressió que es feia a les dades del vídeo eren d'imatge. És a dir, només es realitzava una compressió de cada fotograma del vídeo, sense tenir en compte ni la informació que el precedia o la que s'enviaria a continuació. Cada fotograma individual era comprimit i enviat.

Aquesta manera de compressió és primitiva i, realment, no és un mètode de compressió de vídeo. La compressió de vídeo real, a part de comprimir els fotogrames amb algorismes de compressió com pot ser JPG o JPEG, a més a més involucra compressió temporal. Com el seu nom indica, la compressió temporal actua en la dimensió del temps. Funciona comparant un fotograma fix que conté una imatge sencera, anomenat "keyframe", o fotograma clau, amb els adjacents a aquest i, en lloc de guardar tota la informació de cada imatge, només se salven les diferències entre els fotogrames. D'aquesta manera, els fotogrames successius, anomenats ïnterframes", o fotogrames intermedis, són construïts per sobre del fotograma complet, estalviant tota la informació redundant que no canvia entre imatges.

En la següent figura es pot veure el protocol utilitzat en la compressió temporal MPEG, una de les més populars. En el seu cas, els keyframes són etiquetats amb la lletra Ï", ja que s'anomenen Ïntraframes", mentre que els fotogrames que es construeixen a partir d'aquests es referencien com fotogrames "P", o predictius, on es guarda la diferència entre el fotograma anterior i l'actual, i "B", o bi-predictius, que tenen en compte tant el fotograma precedent com el vinent. L'ús d'aquests fotogrames "B"és opcional.

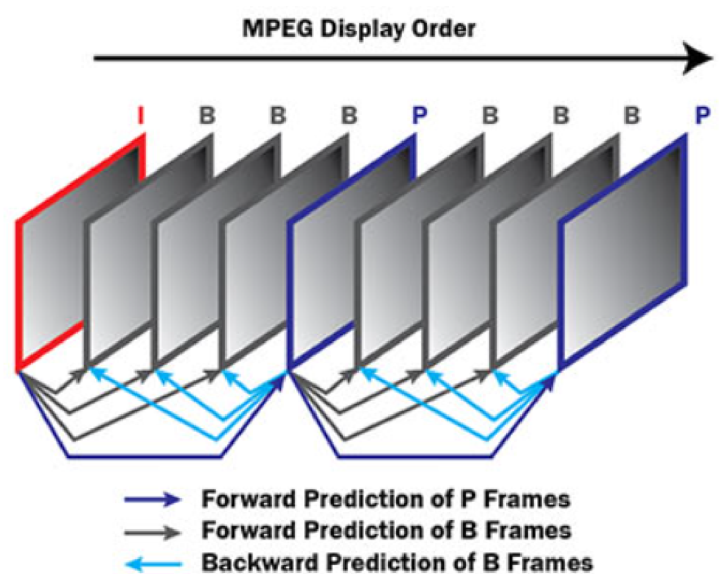

FIGURA 8.9: Fotogrames d'un vídeo comprimit amb L'estàndard MPEG

8.5.3 Compressió Theora

El paquet emprat per convertir el vídeo gravat a un *topic* de ROS, a més a més de proporcionar un tema per la imatge sense comprimir i la imatge comprimida, també oferia un altre *topic* que proporcionava la transmissió comprimida temporalment amb el mètode Theora. Es tracta d'un mètode de compressió d'imatge i temporal que funciona de la manera explicada a la secció anterior. Per tant, es va optar per retransmetre aquest *topic* per compartir el vídeo entre màquines.

L'amplada de banda utilitzat va disminuir dramàticament, i, en intervals esporàdics, es podia captar una transmissió fluida del vídeo. Tot i això, Theora no va acabar sent el mètode de compressió finalment implementat, ja que constava de dos problemes:

1. Abans de començar a enviar el flux de fotogrames al receptor, el publicador de Theora, en establir connexió amb un subscriptor, li envia tres paquets de capçalera per informar de les característiques del vídeo. Sense conèixer aquestes propietats, no és possible reproduir el contingut.

   En una execució de ROS corrent, aquest fet no suposaria cap problema. En el cas del ROS2ROS, en canvi, com que en parlar entre màquines es genera un subscriptor "fantasma"que s'enllaça amb el *topic* de la seva màquina, per tal de comunicar el què rep a l'altre *roscore*, aquests missatges de capçalera només són enviats quan es connecta el subscriptor "fantasma". En el cas que el subscriptor de la màquina remota no estigui operatiu quan s'engega el ROS2ROS a la màquina que transmet el vídeo, o en el cas que es perdi algun d'aquests paquets de capçalera en la comunicació entre *roscores*, el subscriptor remot no serà capaç de reproduir les dades emeses.

2. Degut al *jitter* (oscil·lacions del temps de latència) que poden tenir els Luma, el directe que rep el segon ordinador pot acabar cada veda més endarrerit, amb més latència, ja que no hi ha implementat cap mena de control temporal.

El primer problema es va intentar solucionar implementant una retransmissió periòdica d'aquests missatges de capçalera, però es va descartar perquè emetre els missatges aturava durant uns segons la transmissió de vídeo.

8.5.4 RTSP

La solució que va acabar amb aquests problemes i va aconseguir una transmissió correcta del vídeo va ser integrar, en lloc d'una retransmissió d'un *topic* de vídeo comprimit, un servidor fet exprés per a fluxos de dades en temps real amb compressió integrada.

Un servidor RTSP permet la connexió d'un client en qualsevol moment i, a més a més, consta d'un protocol de control temporal, que ajusta l'enviament a les necessitats del client.

Implementant el servidor del paquet de ROS *ros_rtsp*, el qual transforma un *topic* a un stream RTSP i permet ajustar la compressió del vídeo per assolir l'ús d'amplada de banda desitjat, es va poder compartir el vídeo de prova d'una màquina del CIRS a l'altra a través dels mòdems òptics Luma.

S'explica en més detall el funcionament a la secció 10, i el resultat de la transmissió de vídeo a la secció 9.

Capítol 9

Implementació i Resultats

9.1 Enviament de missatges

9.1.1 Simulació en màquines virtuals

En desenvolupar una versió inicial del paquet ROS2ROS, es va provar d'enviar missatges simples de text a través de *topics* de dues execucions separades de ROS. En la figura 9.1 es veu com, a través del *topic* /send_udp/string, una màquina publica la paraula "test", mentre que l'altre rep el missatge amb el programa ROS2ROS, i amb la comanda *echo* de ROS es mostra com aquest missatge ha estat publicat en el *topic* d'aquesta màquina.

FIGURA 9.1: Procés d'enviament de la paraula "test"a través del *topic* /send_udp/string, inicialitzat amb dos *roscores* diferents i comunicat per ROS2ROS

En la figura 9.2 es mostra com la interacció anterior només ha suposat un missatge d'UDP, com estava implementat.

ip.addr == 8.8.8.5

No.	Time	Source	Destination	Protocol	Length	Info
627	88.656551488	8.8.8.4	8.8.8.5	UDP	52	53344 → 10004 Len=8

FIGURA 9.2: Paquets capturats a la conversa entre les màquines virtuals en el test de la figura 9.1

En la figura 9.3 es mostra com missatges de text més extensos, com un missatge de tipus Odometria, també es passa sense problemes.

Per a la comunicació completa també és necessari habilitar la interacció entre serveis. Per provar-ho, es va implementar un servei simple que respongués a un *Trigger* (un missatge buit que s'envia per solicitar l'acció d'un servei), es va habilitar en una màquina, i a l'altra es va utilitzar la funció *rosservice call* per cridar al servei remot. En la figura 9.4 es pot veure com la petició és resposta sense problemes.

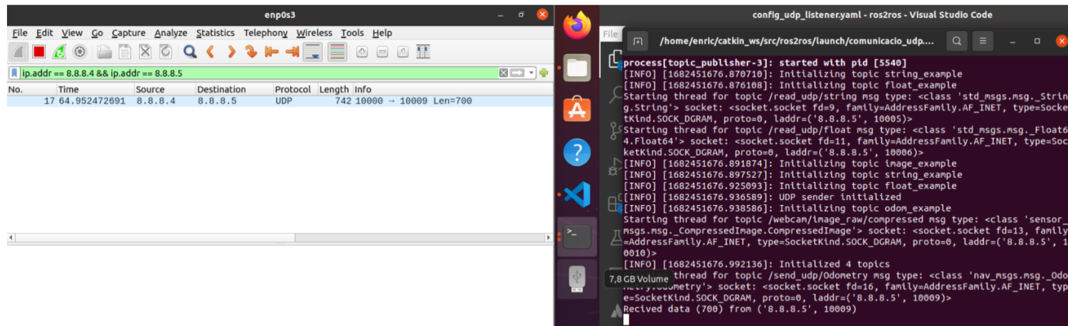

FIGURA 9.3: Pantalles de les màquines virtuals en enviar-se un missatge de tipus Odometria mitjançant el paquet ROS2ROS. La màquina remitent (esquerra) mostra la captura de Wireshark del missatge, mentre que la receptora (dreta) assenyala que ha rebut el missatge

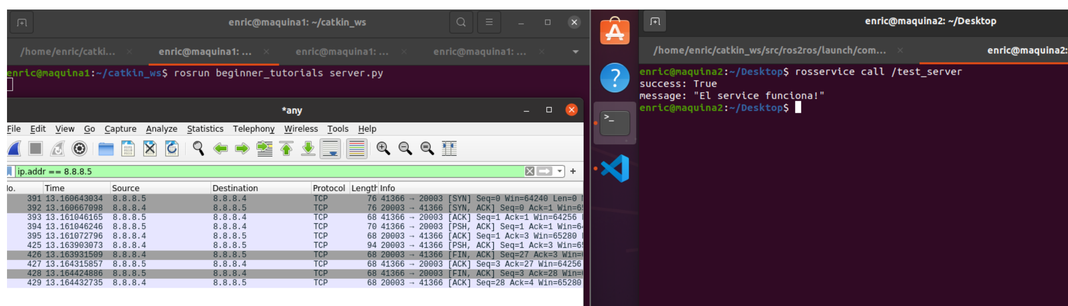

FIGURA 9.4: Interacció d'un servei simple entre màquines virtuals comunicades pel paquet ROS2ROS, junt amb la captura de Wireshark entre ambdues màquines

9.1.2 Primeres proves en els mòdems

Posteriorment, es va deixar la simulació de les màquines virtuals a una banda, i es van repetir les proves anteriors emprant els mòdems òptics Luma. El resultat va ser idèntic.

També es va provar d'enviar paquets més demandants, com ara enviar imatge. En la figura 9.5 es veu com es va aconseguir comunicar *topics* d'imatge entre les màquines del CIRS. Una sola imatge s'enviava sense problemes, però pels fets comentats en l'apartat 8.5.1, intentar visualitzar un *topic* on es retransmet vídeo en imatge era inviable.

9.2 Transmissió de vídeo

Un cop revisat el funcionament correcte del programa per a missatges senzills, es va passar a solucionar el veritable problema de

9.2.1 Saturament de la xarxa

Després de provar diferents mètodes, comentats a la secció 8, es va instaurar un servidor RTSP en el paquet ROS2ROS, el qual permetia una retransmissió controlada en el temps, a la qual es podia accedir en qualsevol moment. Mitjançant un vídeo estandarditzat, es va provar de transmetre'l d'una màquina a l'altre a través dels mòdems submergits. Es va provar el servei de RTSP i, per tenir un punt de contrast

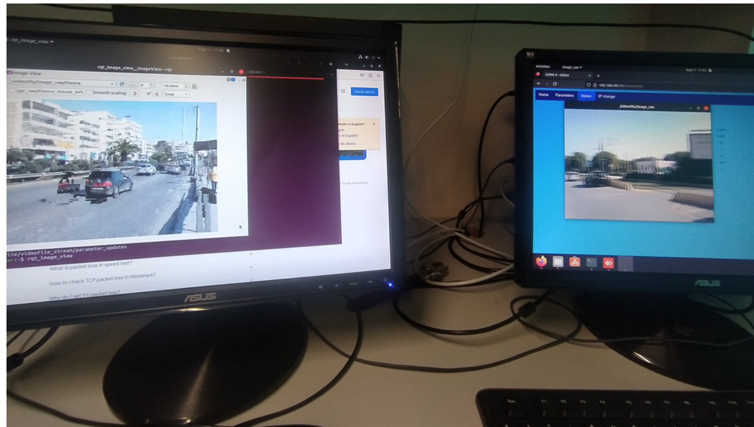

FIGURA 9.5: Pantalles de les màquines reals comunicades pels mòdems òptics, enviant-se imatge per primera vegada

sobre quina diferència feien els protocols instaurats, també es van fer proves usant un sol *roscore*, comunicant les màquines per TCP. Originalment, els resultats van ser els següents:

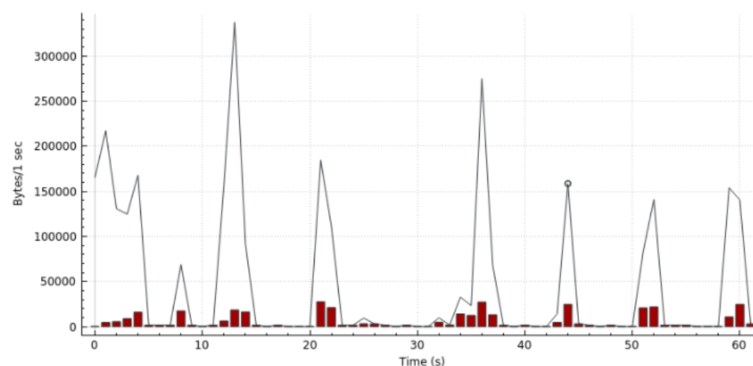

FIGURA 9.6: Paquets capturats en la màquina receptora en la primera prova d'enviament de vídeo a 500 kbits/s per ROS TCP

Com es pot observar a la figura 9.8, el vídeo enviat, comprimit per assolir un enviament de dades 500 kbits/s, resulta en un flux constant que s'aproxima als 200 kbytes/s. Al costat del client, en canvi, es reben només pics d'aquesta informació, amb pèrdues importants entremig, tant per UDP com per TCP.

Els pics de TCP són més estrets, indicant que menys porcions del vídeo són enviades correctament que per UDP, degut a les retransmissions dels paquets perduts, els quals ocupen l'amplada de banda disponible.

Tot i això, aquestes pèrdues revelen un comportament estrany en la xarxa. Si aquesta tingués un percentatge alt de pèrdues, el que rebria la màquina client serien menys dades en general, però un flux igual de constant. En aquest cas es veuen pics on la transmissió és correcte i, sobtadament, es perd.

Dur a terme la mateixa prova a través d'un canal de comunicació sense pèrdues, mitjançant màquines virtuals, resulta en una connexió estable:

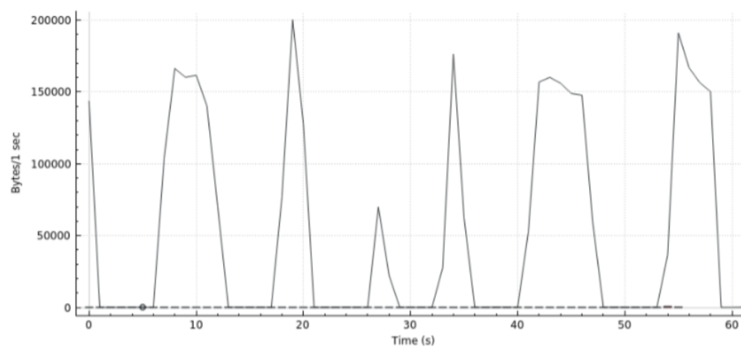

FIGURA 9.7: Paquets capturats en la màquina receptora en la primera prova d'enviament de vídeo a 500 kbits/s per RTSP UDP a través dels mòdems Luma

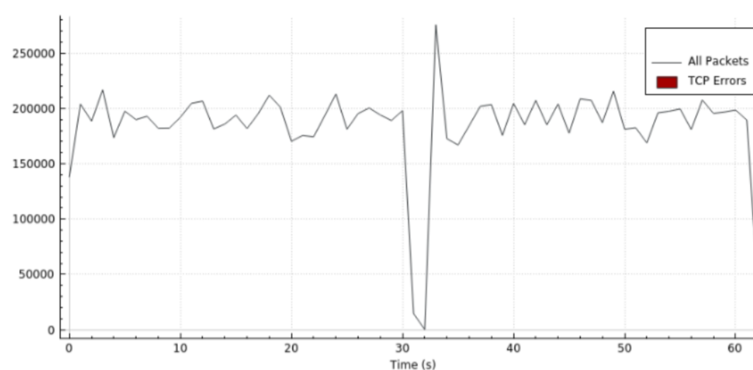

FIGURA 9.8: Paquets capturats en la màquina remitent en la primera prova d'enviament de vídeo a 500 kbits/s per RTSP UDP a través dels mòdems Luma

Proves posteriors delaten que aquestes pèrdues només sorgeixen en enviar una certa quantitat de dades a través dels mòdems òptics.

En la figura 9.10, la primera pèrdua de connexió succeeix en els primers segons, mentre que el punt d'inflexió en la figura 9.11 no arriba fins al segon 50 de l'experiment, on, en lloc de tallar-se la connexió per complet, cada vegada es reben menys paquets.

Aquest comportament s'explica amb la saturació dels mateixos mòdems Luma. La quantitat d'informació que reben és molt major a la que poden processar i retransmetre. En conseqüència, tots els paquets que són enviats mitjançant el mòdem remitent, tot i ser captats pel mòdem receptor, són descartats si aquest últim té la memòria on s'escriuen les dades rebudes plena.

9.2.2 Augment de receptors

Experimentant amb els paràmetres dels mòdems òptics, es va trobar que aquests disposen d'un total de quatre receptors de llum, dels quals, si es deixa al controlador dels mòdems triar com automàticament com es rep la llum, sempre se n'utilitza solament un.

Especificant manualment l'ús de tots els receptors, es pot apreciar una millora notable en la quantitat i qualitat de informació rebuda. Com es pot veure en el gràfic

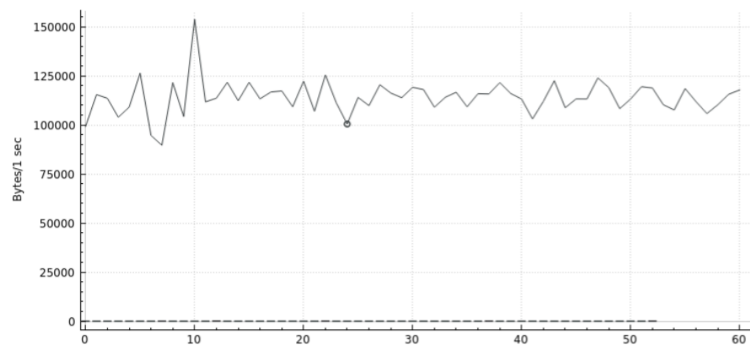

FIGURA 9.9: Paquets capturats per part de la màquina receptora en l'enviament de vídeo a 500 kbits/s per RTSP UDP a través d'una connexió virtual

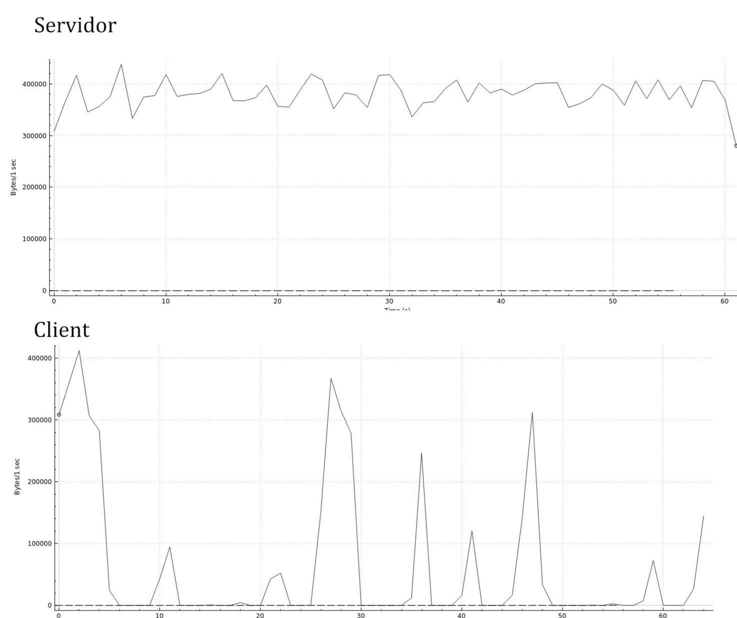

FIGURA 9.10: Paquets capturats en l'enviament de vídeo a 1000 kbits/s per RTSP UDP

capturat de la figura 9.12, l'augment dels receptors permet una retransmissió de pràcticament totes les dades per a bitrates de 500 kbit/s, i és comença a perdre's informació a partir dels 700 kbits/s.

9.2.3 Límit de l'amplada de banda emprat per la màquina

La interfície de Linux, per defecte, les dades que retransmet per Ethernet surten a una velocitat de 100 Mbits/s. Els Luma, en canvi, pel que s'ha observat, amb el software actual només són capaços d'enviar-se 2,5 kbytes/s sense pèrdues, que equivalen a 2 Mbit/s. Aquesta disparitat fa que, si s'intenta enviar un volum major a 2,5 kbytes/s, la majoria dels paquets de més seran descarats.

Mitjançant l'eina de Linux *tc* (Traffic Control), es pot limitar aquesta sortida perquè els valors concordin. Fent aquesta millora, en la figura 9.13 es pot comprovar la sortida de la màquina remitent a l'hora d'enviar vídeo és molt més similar a la què

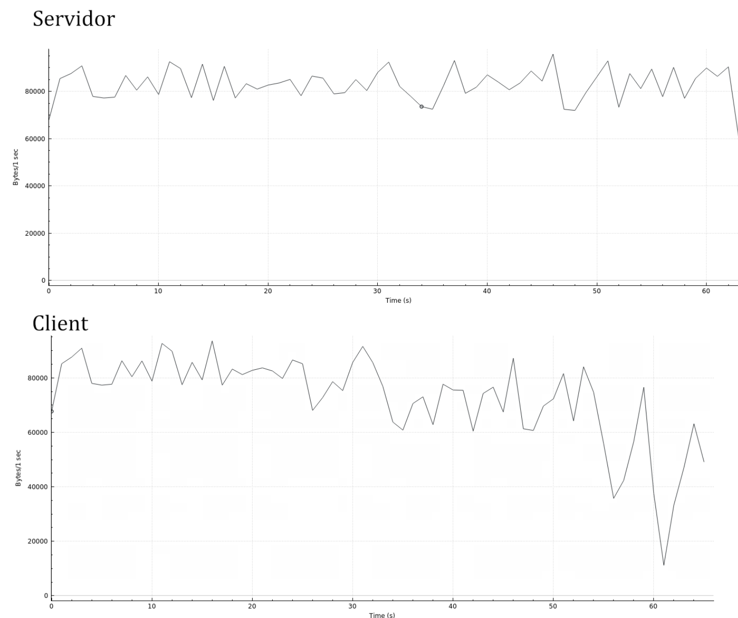

FIGURA 9.11: Paquets capturats en l'enviament de vídeo a 200 kbits/s per RTSP UDP

rep la màquina receptora, arribant al límit d'amplada de banda en enviar bitrates de 700 kbits/s.

D'aquesta manera, les dades que es generin per sobre de l'amplada de banda dels Luma poden ser retingudes en la màquina remitent, en lloc de ser descartades. En aquestes últimes proves, el vídeo rebut era d'una qualitat més que acceptable durant la retransmissió a 500 kbits/s. Per bitrates més alts, el vídeo era funcional, no sorgien artefactes de paquets defectuosos, però el framerate es veia afectat.

9.3 Eina Iperf

Iperf és un programari desenvolupat pel Distributed Applications Support Team (DAST), que permet realitzar proves sobre xarxes informàtiques. Ofereix mesurar el rendiment d'una connexió mitjançant proves tant en TCP com en UDP. El seu funcionament consisteix en una conversa entre una màquina servidor, la qual espera una petició d'un altre dispositiu, i una màquina client, la qual escull quin protocol emprar i tracta d'enviar tantes dades com sigui possible al servidor.

Per estudiar la connectivitat dels Luma, es va utilitzar l'eina Iperf en el canal dels mòdems. Se'n van extreure els resultats de les figures 9.14, 9.15 i 9.16.

El resultat de la prova de TCP mostra com, com a màxim, es poden enviar 3,34 Mbits/s a través dels mòdems òptics, amb pèrdues d'aproximadament 1,2%.

Per UDP només es poden enviar 1,05 Mbits/s a causa de la falta de control de congestió del protocol. Amb TCP es modula l'enviament segons la capacitat de la banda receptora de rebre els paquets. Si els *buffers* estan massa plens, es redueix la velocitat d'enviament. En canvi, UDP no té en compte res d'això, resultant en pèrdues.

Una prova posterior, en la que se'ls hi indica els Luma utilitzar tots els seus receptors

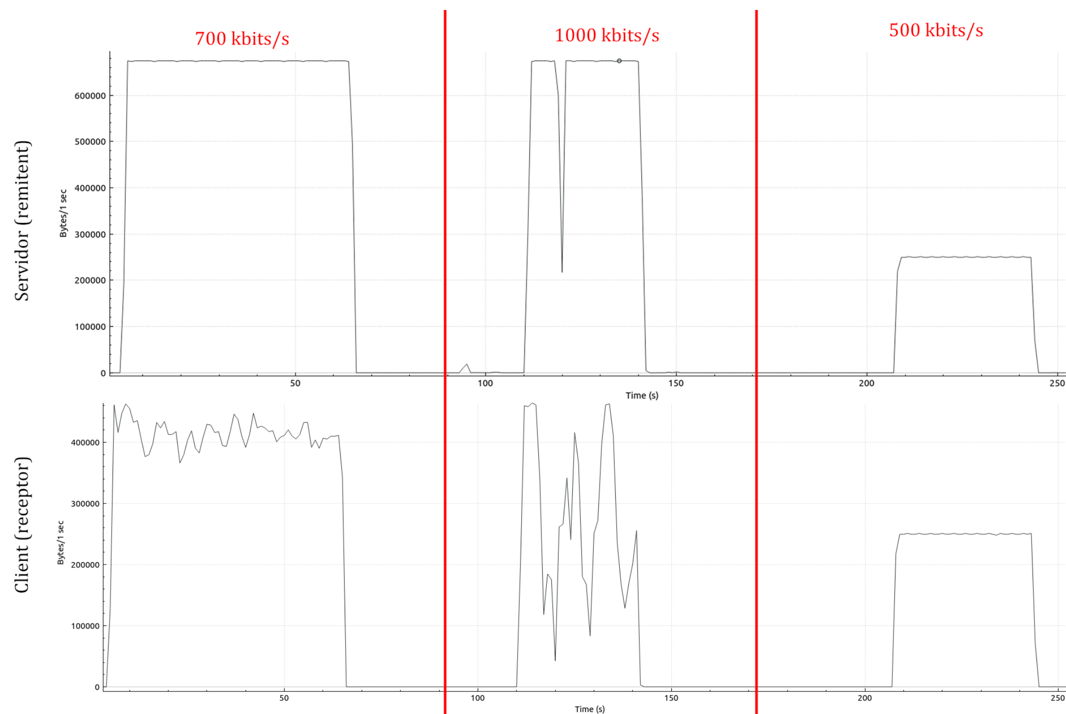

FIGURA 9.12: Bytes capturats a les dues màquines en enviar transmissions de vídeo de diferents bitrates a través dels mòdems òptics amb 4 receptors de llum

de llum, augmentant la mida del *buffer* de missatges, demostra com UDP permet enviar prop de 6,3 Mbit/s, però el client només en rep 4,7 Mbit/s.

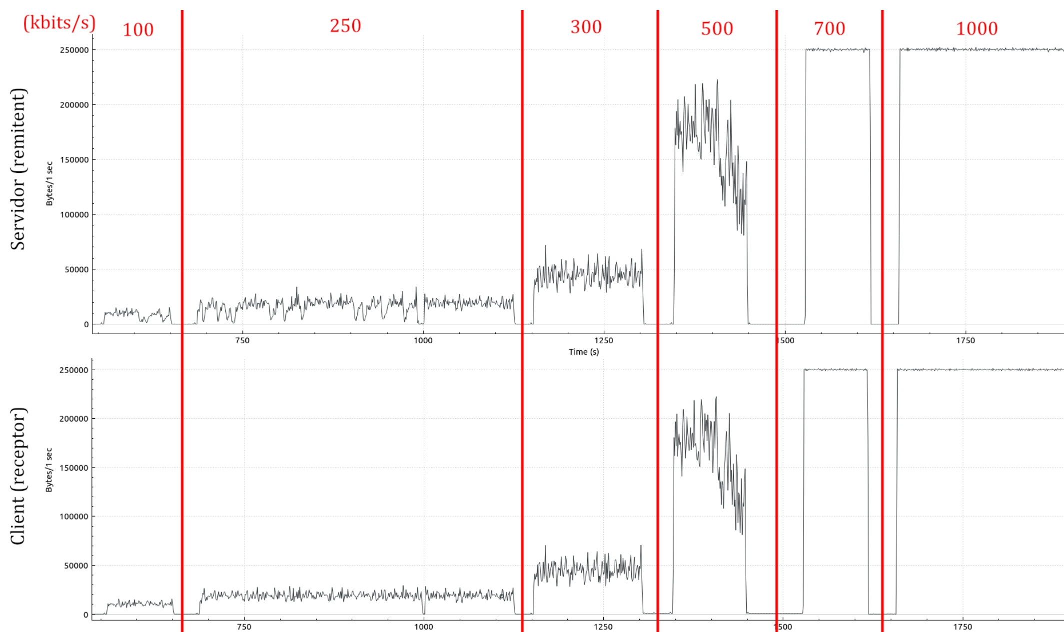

FIGURA 9.13: Bytes capturats a les dues màquines en enviar transmissions de vídeo de diferents bitrates a través dels mòdems òptics amb 4 receptors de llum i l'output limitat

UDP

```
user@user:~$ iperf -c 192.168.102.6 -t 60 -udp
iperf: option requires an argument -- p
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 192.168.102.6, UDP port 5001
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  4] local 192.168.102.8 port 37587 connected with 192.168.102.6 port 5001
read failed: Connection refused
[  4] WARNING: did not receive ack of last datagram after 5 tries.
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-60.0 sec  7.50 MBytes  1.05 Mbits/sec
[  4] Sent 5351 datagrams
```

FIGURA 9.14: Resultat de la prova Iperf per UDP

Server

```
user@computer:~$ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size:  128 KByte (default)
------------------------------------------------------------
[  4] local 192.168.102.6 port 5001 connected with 192.168.102.8 port 38510
[  4]  0.0-60.9 sec  24.2 MBytes  3.34 Mbits/sec
```

Client

```
[  3]  0.0- 2.1 sec  1.12 MBytes  4.51 Mbits/sec
user@user:~$ iperf -c 192.168.102.6 -t 60
------------------------------------------------------------
Client connecting to 192.168.102.6, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.102.8 port 38510 connected with 192.168.102.6 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-60.2 sec  24.2 MBytes  3.38 Mbits/sec
```

FIGURA 9.15: Resultat de la prova Iperf per TCP

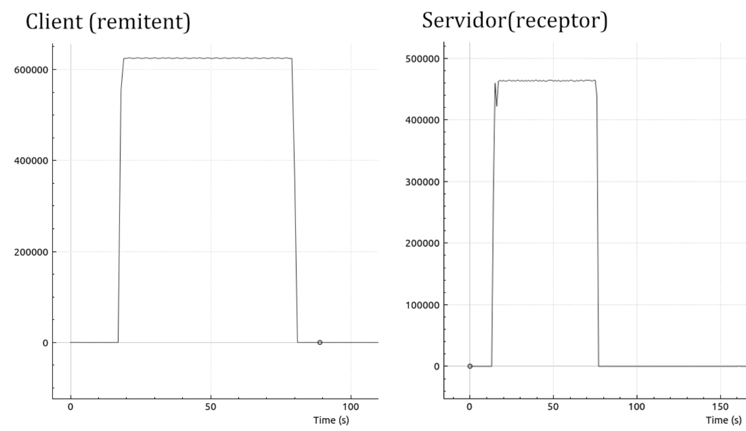

FIGURA 9.16: Resultat de la prova Iperf per UDP amb 4 receptors activats

Capítol 10

Implementació final

El codi complet del paquet es pot trobar penjat al repositori de Github ROS2ROS [5].

10.1 Paquet ROS2ROS

Per tal d'optimitzar l'enviament de dades entre l'operador i el robot, en aquest projecte s'ha optat per separar l'execució de ROS en dos *roscores*, de manera que la màquina operadora i la màquina del submarí utilitzen dues execucions de ROS totalment separades.

Com que es vol mantenir la funcionalitat de comunicació a través dels mètodes subscriptor-receptor i client-servei que proporciona ROS entre les dues màquines, s'ha desenvolupat un paquet de ROS, anomenat ROS2ROS, el qual permet la unificació de dos *roscores* perquè comparteixin certs *topics* i canals de serveis preestablerts.

Correctament configurat, executar el paquet ROS2ROS en dues màquines que usen *roscores* aïllats permetrà que els missatges dels *topics* preestablerts que es publiquin a una màquina apareguin a l'altra, i viceversa. També farà possible contactar amb serveis d'un *roscore* des de l'altre, de manera totalment transparent, com si els tots nodes involucrats estiguessin sota la mateixa execució de ROS.

El funcionament d'aquest paquet s'explica en detall en els següents apartats.

10.2 Codi remitent de *topics* (*sender.py*)

10.2.1 Codi principal

El codi del node per enviar informació d'un *roscore* a un altre comença com tots els altres nodes de ROS: iniciant un node. En aquest cas, s'anomena *ros2ros_sender*.

Tot seguit, s'inicien unes variables que s'usaran més endavant, com un vector que guardarà tots els fils d'execució i l'adreça de la màquina destinatària.

Llavors, es llegeixen els paràmetres de ROS específics per a aquesta aplicació, els quals han d'haver estat carregats prèviament en executar l'arxiu de llançament de la secció 10.8. Primer de tot es llegeix l'adreça de la màquina receptora. Si no estan declarades, salta un error d'execució.

```
145    rospy.init_node('ros2ros_sender')
146
147    threads = []
148
149    # We obtain the target connection's host name
150    host = ""
```

FIGURA 10.1: Primeres línies que s'executen del codi *sender.py*

```
152    if rospy.has_param('sender/address'):
153        host = rospy.get_param('sender/host')
154    else:
155        raise Exception("The parameter 'sender/host' is not set")
```

FIGURA 10.2: Lectura de les adreces del codi *sender.py*

Un cop se saben les adreces amb les que treballar, s'examinen quins són els *topics* que s'han de retransmetre. Només interessa enviar certs *topics* la altra màquina, no és necessari ocupar amplada de banda per dades amb les quals no es preten fer res a l'altra banda de la comunicació. Aquests *topics* també són carregats amb antelació com a paràmetres de ROS en executar l'arxiu de llançament.

En el codi, primer es mira si hi han *topics* especificats. Si n'hi ha, es guarden tots els seus paràmetres en una variable amb el nom de topcis. Altrament, s'avisa l'usuari amb un missatge d'error de ROS.

```
159    # We go through each specified topic to publish
160    if (rospy.has_param('sender/topics')):
161        topics = rospy.get_param('sender/topics')
162
191    else:
192        rospy.logerr("[Sender] The parameter 'topics' is not set")
```

FIGURA 10.3: Condicional sobre els *topics* delcarats com a paràmetre del codi *sender.py*

Amb les dades dels *topics* a l'abast, es recorre cadascun dels temes, per tal d'inicialitzar un subscriptor de ROS específic per a cadascun d'aquests. Dels seus paràmetres se'n llegeixen el nom del *topic* amb el qual s'identifica, el port per on s'enviarà la informació llegida del *topic*, i el tipus de missatge que envia (cada *topic* utilitza el seu propi estil de missatges: text, imatges, arxius en concret, etc...).

També es llegeix el protocol amb el que es desitja enviar la informació captada. La versió del codi a la hora de presentar aquesta memòria suporta enviament per UDP, TCP i RTSP. Per defecte s'enviaran els missatges a través de UDP, si no s'ha especificat en els paràmetres del *topic*.

Per triar quin protocol a utilitzar, cal tenir en compte que, en aquest codi, l'ús d'UDP està dissenyat per enviar paquets petits amb baixa latència, amb la possibilitat de perdre informació, mentre que TCP pot encarregar-se de paquets d'una mida més considerable i amb la possiblitat de retransmissió en el cas que es perdi informació. Depenent de les pèrdues, però, aquesta retransmisió pot suposar una augment exponencial de la latència. El protocol RTSP només està pensat per ser utilitzat únicament com a canal per retransmetre vídeo en directe.

```
for topic in topics:
    rospy.loginfo("[Sender] Initializing topic " + topic)
    if not (rospy.has_param('sender/topics/'+ topic + '/topic') and rospy.has_param('sender/topics/'+ topic
        raise Exception("Every element of the 'topics' parameter must have a 'topic' and a 'type' key")
    topic_name = rospy.get_param('sender/topics/'+ topic + '/topic')
    topic_port = rospy.get_param('sender/topics/'+ topic + '/port')
    topic_type = rospy.get_param('sender/topics/'+ topic + '/type')

    # We check if another protocol is required
    # By default we send the data through UDP
    if rospy.has_param('sender/topics/' + topic + '/protocol'):
        protocol = rospy.get_param('sender/topics/'+ topic + '/protocol')
    else:
        protocol = 'udp'
```

FIGURA 10.4: Lectura dels paràmetres d'un *topic* en el codi *sender.py*

Coneixent tota la informació necessària per crear el subscriptor del *topic*, el següent pas que es realitza és el de crear un fil d'execució nou, el qual executa una funció encarregada de crear el subscriptor de ROS per al *topic* amb el protocol establert. Seguidament, guarda el nou thread al final del vector *threads*.

El control dels *topics* de vídeo enviats per RTSP és administrat pel fitxer explicat a la secció 10.4. Per tant, en la següent figura es veu que, en aquest fitxer, els únics protocols mencionats són els de TCP i UDP.

```
# Create a new thread for each topic
if protocol == 'udp' or protocol == 'UDP':
    threads.append(threading.Thread(target=thread_publish_udp, args=(topic_name, locate(topic_type), host, topic_port)))
elif protocol == 'tcp' or protocol == 'TCP':
    threads.append(threading.Thread(target=thread_publish_tcp, args=(topic_name, locate(topic_type), host, topic_port)))
else:
    raise Exception("Message Protocol " + protocol + " not supported")
```

FIGURA 10.5: Creació dels threads dels subscriptors en el codi *sender.py*

Normalment no és necessari crear un fil d'execució a part per a cada subscriptor, ja que el mateix ROS s'encarrega de crear-lo perquè funcionin simultàniament. En aquest cas, però, pel que fa a al connexió amb protocol TCP, és imprescindible per no aturar la creació dels altres subscriptors (s'explica més en detall en la secció 10.2.4).

De la figura anterior també és important notar l'ús de la funció *locate()*, la qual permet traduir un nom d'objecte de python a l'objecte en si. Això permet declarar subscriptors de qualsevol mena sense haver de modificar el codi.

Per finalitzar el bulce d'inicialització dels *topics*, un cop generat el thread del subscriptor en qüestió, es crida el controlador del thread perquè iniciï la seva execució.

```
# start the thread
threads[-1].daemon = True
threads[-1].start()
```

FIGURA 10.6: Inicialització dels threads generats en el codi *sender.py*

Aquest procés es replica per a cada *topic* indicat als paràmetres preestablerts. Un cop finalitzat el bucle, s'executa la comanda *rospy.spin()* per mantenir el node encés un cop finalitzada l'execució del seu codi.

10.2.2 Fil d'execució UDP

Quan es crea un fil d'execució per generar un subscriptor d'UDP, la funció *thread_publish_udp()* és cridada dintre d'aquest.

El seu funcionament és molt senzill. Primer de tot es crea l'objecte *socket* amb el qual el subscriptor es comunicarà a la màquina destinatària. Es crea amb la opció *DGRAM*, que li indica que el socket que, en lloc de generar una connexió abans de comunicar-se, enviï i rebi datagrames directament. Aquesta opció és necessària pel flux de missatges en UDP.

Addicionalment, a la següent línia es modifica el comportament del *socket* perquè pugui utilitzar una direcció ja assignada a un altre objecte amb la opció *SO_REUSADDR*. Aquesta mesura és afegida per evitar problemes d'adreces ocupades si es reinicia manualment el programa, ja que, normalment, si la execució acaba abrutament, el port no s'allibera fins després d'un lapse de temps.

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

FIGURA 10.7: Inicialització d'un socket UDP en el codi *sender.py*

Amb el *socket* iniciat, l'últim que falta és crear el subscriptor amb la funció integrada de ROS. El següent tall de codi informa per pantalla que s'inicia el subscriptor i crea un subscriptor que escolta al *topic* indicat. Aquest subscriptor, quan rep un missatge del seu *topic*, el processa cridant la funció *helper_callback_udp()*.

```
rospy.logdebug("[UDP sender] Starting thread for " + topic_name)
rospy.Subscriber(topic_name, topic_type, helper_callback_udp(host, port, s))
```

FIGURA 10.8: Inicialització d'un subscriptor UDP en el codi *sender.py*

10.2.3 Callback UDP

La funció *helper_callback_udp()* és una funció auxiliar que permet cridar la funció *callback_udp()* amb un paràmetre de més a davant. Quan un subscriptor rep un paquet del seu *topic*, aquest crida la seva funció assignada, amb el primer paràmetre sent la informació rebuda. Normalment aquestes funcions tenen un sol paràmetre: les dades rebudes. En el cas d'aquest codi, però, interessa també indicar-li a la funció altres paràmetres: l'adreça i port destí, i el socket a utilitzar.

Si el subscriptor cridés la funció *callback_udp()* directament, les dades que es passen com a primer paràmetre ocuparien el lloc de la adreça IP, fent que els paràmetres no concordessin amb els què s'intenten transmetre. Degut a això, és necessari cridar la funció d'aquesta manera, fent servir la funcionalitat *lambda* de Python.

```
return lambda c : callback_udp(c, host, port, socket)
```

FIGURA 10.9: Implementació de la funció *helper_callback_udp()* en el codi *sender.py*

Bàsicament, el que es fa és generar una funció que demana un paràmetre, i, internament, aquesta li dona aquest paràmetre al la primera posició de la funció *callback_udp()*, mentre que les altres posicions estan emplenades per les variables que es volen compartir amb la funció. És a dir, es modifica la funció de callback perquè

admeti només un paràmetre, en lloc de tres, passant-li la informació d'aquests tres abans de ser cridada.

Pel que fa a la funció *callback_udp()* en si, aquesta consisteix de dues parts. Primer es prepara la informació rebuda per ser transmesa a l'altre màquina usant la funció *serialize()*. Aquesta crida converteix el missatge de ROS a una seqüència de uns i zeros, llesta per ser enviada.

```
# Serialize the recived data into a buffer
buff = BytesIO()
data.serialize(buff)
```

FIGURA 10.10: Serialització de les dades rebudes en la funció *callback_udp()* en el codi *sender.py*

El segon pas és, un cop està preparat el paquet per ser enviat, cridar la funció del socket per enviar-lo a la adreça corresponent. Alhora, es notifica per pantalla que s'envia el paquet.

```
rospy.loginfo("[UDP Sender] Sending data (" +str(len(buff.getvalue())) + ") to " + str(host) + ":" + str(port) + " through UDP")
# Data is sent
socket.sendto(buff.getvalue(), (host, port))
```

FIGURA 10.11: Enviament de les dades en la funció *callback_udp()* en el codi *sender.py*

10.2.4 Fil d'execució TCP

La funció de fil d'execució per a TCP comença igual que l'anterior de UDP, creant un socket. Aquesta vegada, però, el socket s'inicia amb l'opció *SOCK_STREAM*, que li indica al socket que s'utilitzarà per crear connexions estables, com les de TCP.

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
rospy.logdebug("[TCP sender] Starting thread for " + topic_name)
```

FIGURA 10.12: Inicialització d'un socket TCP en el codi *sender.py*

Abans de generar el subscriptor, a diferència d'UDP, on les dades s'envien indiscriminadament, amb TCP cal crear un canal de comunicació entre la màquina local i la destinatària. Per fer-ho, s'empra el següent bucle, el qual crida la funció *connect()* del socket referenciat, amb l'objectiu de crear un canal entre les màquines al port indicat. Si la connexió no és acceptada, el bucle informa per pantalla del fet i espera el temps indicat per la variable *wait_time*. Aquesta variable s'augmenta uns quants segons per cada intent per donar el temps que sigui necessari al receptor per començar a escoltar peticions de connexió.

És important treballar amb fils d'execució separats a l'hora de crear cada subscriptor, ja que aquest bucle pot quedar en espera indefinidament, fins que la seva connexió sigui acceptada, bloquejant tots els altres processos del mateix fil d'execució.

Si la creació del canal és exitosa, llavors, de la mateixa manera que en el thread d'UDP, es crida a ROS per generar un subscriptor. Aquest, però, cridarà a la funció *helper_callback_tcp()*.

```
while not connected:
    try:
        rospy.loginfo("[TCP sender] Connecting to " + str(host) + ':' + str(port))
        s.connect((host, port))
        connected = True
    except Exception as e:
        rospy.logwarn("[TCP sender] Failed to connect to " + str(host) + ':' + str(port) + '. Reattempting in ' + str(wait_time)
        time.sleep(wait_time)
        wait_time += 3
```

FIGURA 10.13: Creació del canal de connexió TCP en el codi *sender.py*

10.2.5 Callback TCP

La funcionalitat de callback pels subscriptors que funcinonen en TCP és pràcticament idèntica a la dels que fan servir UDP. La funció *helper_callback_tcp()* realitza la mateixa feina de traduir la funció principal *callback_tcp()* perquè utilitzi un sol paràmetre, i aquesta última funciona de la mateixa manera que la seva contrapart.

L'única diferéncia és el fet que, com que el socket treballa amb un canal de comunicació preestablert, en lloc de llençar paquets al buit, la funció d'enviament del socket que s'ha de cridar és la de *sendall()*, en lloc de la de *sendto()*.

10.3 Codi receptor de *topics* (*receiver.py*)

10.3.1 Codi principal

El bloc principal de codi d'aquest fitxer és molt similar al *main* del arxiu que genera els subscriptors d'enviament, explicat en la secció 10.2.1. Es llegeix la informació que proporiconen els paràmetres de ROS de la mateixa manera, i també es creen els subscriptors segons els mateixos paràmetres. En lloc de subscriptors, però, en aquest cas es construeixen publicadors, els quals fan la funció inversa. En lloc d'enviar a una altra màquina la informació que llegeixen d'un *topic*, processen els paquets que els hi arriben d'una altra màquina i la publiquen a un *topic*.

Les úniques diferències en aquesta secció de codi dignes de menció són el fet que sí que existeix un thread per tractar els missatges de RTSP, i el fet que el publicador es crea abans d'iniciar el nou fil d'execució.

```
199        # create a publisher for the topic
200        pub = rospy.Publisher(topic_name, locate(topic_type), queue_size=10)
201
202        # Create a thread for the topic with the specified protocol
203        if protocol == 'udp' or protocol == 'UDP':
204            threads.append(threading.Thread(target=thread_republish_udp, args=(address, topic_port, pub, locate(topic_type))))
205        elif protocol == 'rtsp' or protocol == 'RTSP':
206            if rospy.has_param('receiver/topics/' + topic + '/rtsp_url'):
207                rtsp_url = rospy.get_param('receiver/topics/' + topic + '/rtsp_url')
208                threads.append(threading.Thread(target=thread_republish_rtsp, args=(rtsp_url, pub)))
209            else:
210                rospy.logerr("RTSP stream has no URL")
211        elif protocol == 'tcp' or protocol == 'TCP':
212            threads.append(threading.Thread(target=thread_republish_tcp, args=(address, topic_port,pub, locate(topic_type))))
213        else:
214            raise Exception("Message Protocol " + protocol + " not supported")
```

FIGURA 10.14: Inicialització del publicador i dels threads en el codi *receiver.py*

10.3.2 Fil d'execució UDP

El thread que genera els publicadors que llegeixen de UDP comença de manera similar a la seva contrapart del arxiu *sender.py*, creant un socket per a UDP. Cal destacar,

però, que a aquest li assignem un port específic amb la funció *bind()*, ja que es distingirà quina informació pertany a quin *topic* segons el port per on arribi.

```
55        # UDP socket that we bind to the address and port
56        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
57        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1
58        s.bind((address, topic_port))
```

FIGURA 10.15: Inicialització d'un socket UDP en el codi *receiver.py*

Llavors, s'inicia un bucle constant, on continuament es llegeix del socket i es republica la informació rebuda en un *topic*.

Dins del bucle, primer s'utilitza la funció *recvfrom()* del socket, la qual es queda escoltant en el port definit del socket, fins que rep alguna cosa.

```
63        # Wait for data
64        data, addr = s.recvfrom(UDP_BUFFER_SIZE)
65        rospy.loginfo("[UDP Receiver] Recived data (" +str(len(data)) + ") from " + str(s.getsockname()) + " through UDP")
```

FIGURA 10.16: Recepció de dades UDP en el codi *receiver.py*

Un cop s'ha rebut un paquet, s'intenta reconstruir el missatge original segons la variable *topic_type*, que indica a quina classe pertanyen els missatges rebuts. Si els missatges rebuts no són originalment provinents de missatges d'aquesta classe, la deserialització fallarà i es notificarà al usuari.

Si és exitosa, però, es crida el publicador del *topic* perquè pengi el missatge, completant la transacció.

```
# Process data
msg = msg_type()
try:
    msg.deserialize(data)
    publisher.publish(msg)
except:
    rospy.logwarn("[UDP Receiver] Couldn't deserialize the data recived from " + str(s.getsockname()))
```

FIGURA 10.17: Deserialització i publicació de les dades rebudes en UDP en el codi *receiver.py*

10.3.3 Fil d'execució TCP

El fil d'execució per crear els publicadors que llegeixen de TCP segueix la mateixa estructura que l'anterior. De la mateixa manera, comença declarant un socket amb la parella adreça-port d'on ha de llegir les dades. Com que es pretén una connexió TCP, cal iniciar el socket amb l'opció *SOCK_STREAM*.

A continuació, igual que en el thread anterior, s'entra en un bucle infinit que escolta pel socket. A diferència de l'anterior, però, en aquest thread primer cal establir un canal de comunicació entre les màquines abans de rebre paquets. Per fer-ho, es crida la funció *listen()*, la qual escolta per a peticions de connexió que pot rebre la màquina en el port indicat pel socket. Un cop es rep una petició, s'accepta amb la funció *accept()* i, llavors, es pot procedir a llegir informació que vingui del canal establert.

```
87      # TCP socket that we bind to the address and port
88      s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
89      s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
90      s.bind((address, topic_port))
```

FIGURA 10.18: Creació d'un socket TCP en el codi *receiver.py*

```
# Listen for incoming conneciton
s.listen(1)
conn, addr = s.accept()
```

FIGURA 10.19: Espera de connexions TCP en el codi *receiver.py*

Per acomplir aquesta tasca, s'entra en un bucle que, mentre el canal estigui obert, llegeix el paquet de la cua a llegir amb la funció del socket *recv()*, i la processa de la mateixa manera que en el thread UDP de l'apartat anterior.

```
101             # Wait for data
102             data = conn.recv(TCP_BUFFER_SIZE)
103
104             if data:
105                 rospy.loginfo("Recived data (" +str(len(data)) + ") from " + str(s.getsockname()) + " through TCP")
106
107                 # Process data
108                 msg = msg_type()
109                 try:
110                     msg.deserialize(data)
111                     publisher.publish(msg)
112                 except:
113                     rospy.logwarn("Couldn't deserialize the data recived from " + str(s.getsockname()))
```

FIGURA 10.20: Desarialització i publicació de les dades rebudes en TCP en el codi *receiver.py*

Si les dades llegides tenen mida zero, vol dir que el canal s'ha tancat. Llavors se surt del bucle i es notifica que la connexió s'ha perdut. Com que aquesta secció està dins d'un bucle sense fi, es torna a l'inici i es torna a escoltar el socket per a una nova connexió.

10.3.4 Fil d'execució RTSP

Si el fil d'execució que s'ha generat intenta llegir un vídeo en directe per RTSP, el codi que s'executarà no és tant similar al dels threads de UDP i TCP.

Per començar, com que es llegirà de una url proporcionada amb antelació als fitxers de configuració, en lloc d'esperar una connexió inicial de l'altre màquina, no és necessari declarar un nou socket. El primer que es fa en aquest apartat és delcarar unes variables de configuració per a la llibreria de *OpenCV*, la qual serà l'encarregada de fer la comunicació per RTSP amb la màquina veïna.

En concret, se li especifica que, per a l'ús de la llibreria *FFMPEG* (llibreria interna de OpenCV que tracta la compresió i captura de vídeo) fagi servir el protocol de transport RTSP per sobre de UDP.

RTSP normalment funciona sobre TCP, però com que es vol prioritzar la baixa latència per sobre de la correcte recepció de tots els paquets, s'indica que s'usi sobre UDP, que no retransmet dades.

```
127     # Set FFMPEG capture options
128     os.environ["OPENCV_FFMPEG_CAPTURE_OPTIONS"] = "rtsp_transport;udp"
```

FIGURA 10.21: Configuració de OpenCV en el codi *receiver.py* per llegir RTSP

El següent pas que es porta a terme és el d'entrar en un bucle sense fi, on es connectarà a la direcció especificada i es llegiran tots els frames del vídeo en qüestió.

En entrar, el primer que es fa és obrir una captura de vídeo amb la llibreria OpenCV, indicicant que s'usi el mode de captura FFMPEG.

```
132     while True:
133         cap = cv2.VideoCapture(address, cv2.CAP_FFMPEG)
```

FIGURA 10.22: Inicialització de la captura del vídeo en directe en el codi *receiver.py*

Si la captura es pot obrir, s'entra en un nou bucle on, mentre la transmissió segueixi activa, es llegeixen els *frames* del vídeo amb la funció *read()* de la captura de OpenCV, es transformen a missatges de ROS de tipus *sensor_msgs/Image* mitjançant la funció *cv2_to_imgmsg()*[6], i són publicats al seu *topic* corresponent d'imatges amb el publicador referenciat.

```
134         while(cap.isOpened()):
135             if not connected:
136                 rospy.loginfo("Reading stream from " + address)
137                 connected = True
138             try:
139                 # Read frame of the video
140                 ret, frame = cap.read()
141 
142                 # Convert frame to sensor_msgs/Image message
143                 frame = cv2_to_imgmsg(frame)
144 
145                 # Publish the frame
146                 publisher.publish(frame)
147             except:
148                 cap.release()
```

FIGURA 10.23: Lectura dels fotogrames del *stream* RTSP rebut en el codi *receiver.py*

Si la funció *read()* falla, es suposa que el canal de RTSP s'ha tancat i es surt del bucle.

Fora d'aquest bucle, si hi ha hagut una connexió establerta, es comunica al usuari que s'ha perdut. Si no, s'escriu que obrir el canal no ha estat possible.

Finalment, el bucle infinit arriba al final del seu bloc i torna a començar per tornar a llegir una connexió RTSP.

10.4 Enviament RTSP

Per retransmetre un vídeo en directe, com per exemple el d'una càmera del robot en operació, el paquet ROS2ROS també suporta la connexió per RTSP (Real Time Streaming Protocol) per a *topics* de ROS de tipus *Image*.

Aquesta funcionalitat ha estat possible gràcies a la incorporació del paquet de ROS *ros_rtsp* al codi, cortesia de l'usuari CircusMonkey, qui va penjar aquest paquet al domini públic. El codi font d'aquest paquet està escrit en C++, a diferència de la resta del programa ROS2ROS, que està redactat completament en Python. Afortunadament, ROS permet l'ús simultani de codi amb diferents llenguatges en el mateix paquet.

El funcionament d'aquest tall de codi consisteix a llegir un flux de vídeo en directe especificat l'arxiu de configuració, ja sigui la sortida d'un *topic* de tipus Imatge (el que es fa servir per a aquest projecte) o la sortida d'una càmera, i crear un servidor capaç de crear connexions RTSP, el qual retransmet el vídeo en qüestió. Es permeten retransmetre múltiples streams a la vegada, sempre que estiguin declarats en el fitxer de configuració comentat en la secció 10.7.1.

Per fer funcionar aquest servidor es fa ús de la llibreria *G-streamer*, la qual permet, no només generar el servidor RTSP amb l'adreça desitjada, sinó que també comprimeix el vídeo, utilitzant l'estàndard de la llibreria MPEG H.264, per tal d'obtenir l'ús d'amplada de banda desitjat.

10.5 Codi client de serveis per TCP (*tcp_client*)

10.5.1 Codi principal

Si un client que funciona en un *roscore* pretén fer una petició a un servei que pertany a una altra execució de ROS, és imperatiu que el codi *tcp_client.py* estigui en execució per la banda de la màquina client.

Aquest codi comença, com la resta de nodes de ROS, inicialitzant el node, aquest cas sota el nom de "tcp_service_client", i llegint els paràmetres adients, com poden ser l'adreça on es troben els serveis a preguntar, i els noms, tipus, i ports dels serveis.

Seguidament, coneixent quins serveis es pretén cridar, s'entra en un bucle per recórrer'ls i, per cadascun d'ells, es crea un "servei fantasma", al qual els nodes locals podran preguntar per fer la petició al servei de la màquina remota. S'inicialitzen amb la funció de ROS *rospy.Service()*.

```
95          srv = rospy.get_param('service_client/services/'+ service + '/service')
96          port = rospy.get_param('service_client/services/'+ service + '/port')
97          topic_type = rospy.get_param('service_client/services/'+ service + '/type')
98
99          # Initialize service
100         rospy.Service(srv, locate(topic_type), helper_callback(address, port, locate(topic_type)))
```

FIGURA 10.24: Inicialització d'un servei en el codi *tcp_client.py*

De la manera similar a l'inici d'un subscriptor, els serveis se'ls hi dona el canal per on rebran informació, el tipus de dades que hauran de processar, i una funció çallback", la qual accediran en rebre una petició d'un client.

Un cop declarats tots els serveis a preguntar, es crida la comanda *rospy.spin()* per mantenir-los oberts un cop s'acabi l'execució del codi.

10.5.2 Callback

Els serveis d'aquest codi, quan reben una petició, la processen usant la funció *helper_callback()*, funció auxiliar explicada en la secció 10.2.3 per poder cridar la verdadera funció objectiu, la funció *callback()*.

La feina d'aquesta funció consisteix a enviar la petició rebuda al servei remot, i recollir la seva resposta, de la següent manera:

Primer de tot es declara un socket per a connexions TCP com a la figura 10.12 i, seguidament, s'intenta fer una connexió amb el servidor remot. A diferència de l'enviament per TCP dels *topics*, es genera una connexió nova per a cada petició, ja que aquestes són puntuals, en lloc de la comunicació constant dels *topics*.

Un cop establerta la connexió, es serialitza el missatge rebut del client per tal de ser enviat cap a l'altra màquina. A més a més, se li afegeixen dos bytes nuls al darrere. Un per evitar errors de missatges buits, ja que els serveis poden tenir peticions nul·les com ara les de tipus *Trigger*, i un altre per indicar el fi del missatge.

Llavors, s'envia la petició amb la funció del socket connectat *sendall()* i s'espera que el servei receptor retorni una resposta amb la funció *recvfrom()*. Si s'ha rebut una resposta exitosament, es desserialitza segons el tipus d'objecte especificat en la configuració, ja que la resposta també ve empaquetada com a bits dins d'una transmissió TCP.

```
41      data = s.recvfrom(4096)
42      rospy.logdebug("[TCP Service Client] Received response")
43      if not data:
44          rospy.logerr("[TCP Service Client] No data received! If empty response, this might not be handled!")
45
46      # Process response
47      res = srv_type._response_class()
48      try:
49          res.deserialize(data[0])
50      except Exception as e:
51          rospy.logerr("[TCP Service Client] Error while deserializing response: " + str(e))
52          res = None
```

FIGURA 10.25: Recepció de la resposta del servei remot en el codi *tcp_client.py*

Per acabar, es talla la connexió amb el servei remot tancant el socket amb la funció *shutdown()* i es retorna la resposta al client original.

10.6 Codi servidor de serveis per TCP (*tcp_client*)

10.6.1 Codi principal

Pel que fa a la banda del codi que tracta les peticions dels clients remots, la primera secció del codi, com ja s'ha vist en els altres executables, comença declarant un node i llegint els paràmetres de la configuració carregats al *roscore*.

Sabent quins serveis se'ls hi ha d'habilitar la comunicació amb el *roscore* remot, es procedeix a generar un socket per cadascun d'aquest, de manera que es diferencien les peticions entrants pel port on arriben. Tot seguit, es declara també un objecte de

tipus *ServiceProxy*, el qual permet fer peticions, també conegut com fer "proxy", al servei local que s'especifica en la variable *srv* en crear-lo.

```
102         # Create the TCP socket
103         s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
104         s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
105         s.bind((address, port))
106
107         # Create the proxy object of the service
108         prox = rospy.ServiceProxy(topic, locate(topic_type))
```

FIGURA 10.26: Inicialització d'un socket TCP i un objecte *ServiceProxy* en el codi *tcp_service.py*

Com que és necessari escoltar a través de múltiples sockets a la vegada, un per cada servei local a comunicar, es crea un fil d'execució separat per cadascun d'aquests, als quals se'ls dona el seu respectiu socket, tipus de servei, i objecte "proxy"enllaçat amb el servei local, i s'inicia immediatament.

```
110         # Create a thread for the socket
111         th = threading.Thread(target=thread_server, args=(prox, locate(topic_type), s))
112
113         # Start the thread
114         th.daemon = True
115         th.start()
```

FIGURA 10.27: Inici dels fils d'execució del codi *tcp_service.py*

10.6.2 Fil d'execució del servidor

Cada servei local que es comunica amb l'exterior ho fa mitjançant un fil d'execució a part, el qual comença esperant per una connexió al socket que se li ha brindat i escoltant el què es rep pel canal amb la funció de socket *recv()*.

Un cop arriba una petició d'algun client, es desserialitza, encaixant la informació rebuda a la classe de missatge especificat a la configuració. Si el missatge és buid es considera que s'ha acabat la transmissió i es talla la connexió. Si no es pot desserialitzar, salta un error i es notifica a l'usuari.

Llavors, coneixent la intenció del client, es pregunta al servei local, mitjançant la funció *call()* de l'objecte enllaçat amb el servei que s'ha creat al codi principal. En rebre la resposta, aquesta és serialitzada per ser enviada al client original, afegint un byte nul al final com a indicació de final de missatge.

```
52          # The translated request is sent to the local service
53          res = client.call(req)
54
55          # Serialize the response
56          buff = BytesIO()
57          res.serialize(buff)
58          buff.write(struct.pack('B', 0)) # add a null byte to the end of the message
```

FIGURA 10.28: Petició a un servei local del codi *tcp_service.py*

Per acabar, s'envien les dades codificades amb la funció de socket *sendall()* i, un cop enviada tota la resposta, es tanca la connexió TCP amb la funció *close()*.

10.7 Arxius de configuració

Per tal d'indicar als nodes de ROS d'aquest paquet quins mètodes usar per parlar-se, existeix una carpeta a part del codi principal anomenada 'config' amb una sèrie de fitxers de configuració els quals dicten el comportament dels primers.

El contingut d'aquests fitxers de extensió *.yaml* són carregats com a paràmetres de ROS en executar els fitxers d'execució que els contenen, i els programes que els requereixen els poden llegir en qualsevol moment amb les funcions *rosparam.has_param()* (per consultar si un paràmetre existeix) i *rosparam.get_param()* (per aconseguir-ne el seu valor).

Pel correcte funcionament del progama, cada màquina ha de tenir els seus fitxers de configuració actualitzats segons la informació que es pretén compartir.

10.7.1 Configuració de la retransmissió de *topics*

Per determinar els paràmetres que faran servir els nodes encarregats de comunicar els canals públics entre *roscores*, existeixen tres arxius diferents de configuració:

- config_receiver.yaml
- config_sender.yaml
- stream_setup.yaml

El primer arxiu, el *config_receiver.yaml*, indica al programa quina mena de dades espera llegir dels ports de la màquina local, per llavors retransmetre-les pels *topics* indicats. Es diferencia quin *topic* es llegeix segons el port pel qual n'arriben els seus paquets. Com es pot veure a la figura anterior, el fitxer de configuració del receptor

```
1  receiver:
2    address: '8.8.8.4' # Local address
3
4    topics:
5      octomap:
6        topic: "/octomap_point_cloud_centers"
7        type: "sensor_msgs.msg.PointCloud2"
8        port: 10006
9        protocol: "tcp"
```

FIGURA 10.29: Exemple de configuració del node receptor de *topics*

es divideix en seccions, o a cadascuna es poden trobar diferents paràmetres. Primer es determina quina adreça ha d'utilitzar el programa per rebre les dades de l'altre *roscores*, i llavors segueix una secció on es determinen quins *topics* s'espera rebre.

Per cada *topic* cal especificar quin és el seu nom pel qual l'identificaran els nodes (la variable "*topic*:"), el tipus de missatges el qual utilitzarà aquest *topic*, per poder deserialitzar-los un cop rebuts pel socket i per determinar quina mena de publicadors s'ha d'usar per penjar dades al *topic*, i, per últim el port per on s'espera rebre els paquets i amb quin protocol seran enviats. Els protocols soportats en aquesta versió del codi són TCP, UDP i RTSP. L'últim s'usa únicament per a vídeo en directe, i cal proporcionar la url del flux de dades enlloc del port i el tipus de *topic*.

També cal destacar que a la primera línia s'escriu "receiver:", així tots els paràmetres d'aquest arxiu, un cop carregats, el seu nom precedeix del la paraula "/receiver", per poder distingir-los correctament.

Els altres fitxers de configuració proporcionen la configuració per la part remitenta del programa. El fitxer *config_sender.yaml* funciona idènticament que al primer explicat. Les úniques diferències són el fet que també conté l'adreça de la màquina destintaria, en lloc de la local, ja que només cal conèixer a quina direcció transmetre els paquets, i que només determina la configuració de les connexions per UDP i TCP.

Les connexions per RTSP d'imatge són configurades per l'ultim fitxer, el *stream_setup.yaml*. En aquest arxiu s'especifica a quina adreça es generaran els *streams*, seguint el format de "rtsp://localhost:port/mountpoint", on *localhost* és l'adreça de la màquina, i el port i el mountpoint són especificats al fitxer. Es poden fer varies retransmissions alhora, totes sota el mateix port, però a diferents mountpoints. Aquest *.yaml* també declara, per cada directe, quina és la seva font i el tipus d'aquesta (en el cas d'aquest programa sempre serà de tipus "*topic*"), les restriccions del vídeo que es retransmet, com ara les dimensions o la velocitat dels fotogrames, i, per acabar, el *birate* o ús d'ample de banda que es vol aconseguir. Si el vídeo és massa pesant per aconseguir el *bitrate* desitjat (normalment ho és, ja que es parteix de imatges sense comprimir), el programa s'encarrega de comprimir-lo fent servir l'estàndard H.264 de la família MPEG.

```
5   port: "8554"
6   streams:
7
8     stream-1:
9       type: topic
10      source: /videofile/image_raw # Image rostopic. It must be raw image! Don't use transport topics!
11      mountpoint: /back
12      caps: video/x-raw,framerate=10/1,width=640,height=480 # Constraints of the video stream
13      bitrate: 500 # Target bitrate of the stream
```

FIGURA 10.30: Exemple de configuració del node remitent de *topics* per a les comunicacions en RTSP

10.7.2 Configuració de la retransmissió de serveis

Pel que fa a la comunicació entre nodes de tipus *service* i els seus clients, existeixen dos fitxers de configuració:

- config_tcp_client.yaml
- config_tcp_service.yaml

Aquests fitxers funcionen de la mateixa manera que els arxius *config_receiver.yaml* i *config_sender.yaml* vists en l'apartat anterior. El que acaba en çlient"determina a quins serveis de la màquina contrària es poden fer peticions, mentre que el que acaba en "serviceçomunica els serveis locals amb els clients de l'altre costat.

Com es pot veure en la següent figura, la configuració segueix el mateix format que la configuració dels *topics*, tret de que, en lloc d'especificar *topics*, s'especifica serveis, i que no cal especificar el protocol, ja que les comunicacions amb serveis sempre es faran per TCP.

```
1 service_client:
2   address: '8.8.8.5' # Target address
3
4   services:
5     string_example:
6       service: "/disable_thrusters"
7       type: "std_srvs.srv.Trigger"
8       port: 20003
```

FIGURA 10.31: Exemple de configuració del node client de serveis

10.8 Fitxers d'execució

10.8.1 Fitxers *.launch*

La manera usual d'executar codi en ROS és cridar per la consola la comanda *rosrun* seguida del nom del paquet i el nom de l'arxiu dins d'aquest que es vol iniciar. Aquesta comanda, però, només permet engegar un únic fitxer cada vegada, i, a més a més, necessita que un *roscore* ja estigui funcionant.

Per executar funcionalitats de paquets de ROS més complexes, per evitar haver de fer nombroses crides, existeixen els fitxers amb extensió *.launch*, els quals viuen en una carpeta a part del codi font i condensen totes les crides que es poden fer en una sola comanda.

Per executar un fitxer *.launch*, l'únic que cal realitzar és una crida per consola a la funció *roslaunch*, seguida del paquet que conté l'arxiu i el nom d'aquest. En executar la crida, un *roscore* s'iniciarà automàticament si no n'hi ha cap funcionant, i, seguidament, es carregaran tots els arxius especificats dins de l'arxiu *.launch*, en l'ordre que estiguin col·locats dins d'aquest.

10.8.2 Fitexer d'execució de la comunicació per *topics*

Per permetre la comunicació per *topics* entre dues màquines amb *roscores* separats, cal llençar a cada màquina l'arxiu d'execució *comunicacio_topics.launch*. Aquest arxiu s'encarrega de fer tres coses diferents.

Primer de tot, es carreguen com a paràmetres de ROS els arxius de configuració de la comunicació dels *topics* amb la comanda *load* de ROS.

```
2     <!-- Load the sender/receiver parameters -->
3     <rosparam command="load" file="$(find ros2ros)/config/config_receiver.yaml" />
4     <rosparam command="load" file="$(find ros2ros)/config/config_sender.yaml" />
```

FIGURA 10.32: Càrrega dels paràmetres de configuració del codi *comunicacio_topics.launch*

Seguidament, ara que ja estan tots els paràmetres carregats, s'executen el node de recepció i el node d'enviament, sota el nom de *topic_listener* i *topic_publisher* respectivament. La comanda *output=screen* indica que tot el text que puguin generar, es mostri per a la finestra on s'ha cridat l'arxiu d'execució.

Per acabar, també s'inicialitza el node que s'encarrega de l'enviament de vídeo per RTSP. Aquest, però, com que també necessita iniciar altres arxius per funcionar, té el seu propi arixu d'execució, el qual és cridat.

```
6       <!-- Start the reciver and sender nodes -->
7       <node name="topic_listener" pkg="ros2ros" type="receiver.py" output="screen"/>
8       <node name="topic_publisher" pkg="ros2ros" type="sender.py" output="screen"/>
```

FIGURA 10.33: Crida a la execució dels nodes de comunicació per *topics* del codi *comunicacio_topics.launch*

```
10      <!-- Start the rtsp node -->
11      <include file="$(find ros2ros)/launch/rtsp_streams.launch"/>
```

FIGURA 10.34: Crida a l'execució de l'arxiu que llença el servei d'enviament de vídeo RTSP del codi *comunicacio_topics.launch*

10.8.3 Fitexer d'execució de la comunicació per serveis

Si la comunicació que es vol permetre és entre serveis de ROS, només s'ha de cridar l'arxiu d'execució *comunicacio_serveis.launch*.

Aquest, de la mateixa manera que l'arxiu d'execució anterior, primer carrega uns arxius de configuració *.yaml* i, seguidament, executa els nodes receptor i remitent corresponents.

```
1  <launch>
2      <!-- Load the client and service parameters -->
3      <rosparam command="load" file="$(find ros2ros)/config/config_tcp_client.yaml" />
4      <rosparam command="load" file="$(find ros2ros)/config/config_tcp_service.yaml" />
5
6      <!-- Start the nodes -->
7      <node name="service_serviceside" pkg="ros2ros" type="tcp_service.py" output="screen"/>
8      <node name="service_clientside" pkg="ros2ros" type="tcp_client.py" output="screen"/>
9  </launch>
```

FIGURA 10.35: Codi de l'arxiu *comunicacio_serveis.launch*

10.8.4 Fitexer d'execució de l'aplicació d'enviament de paquets RTSP

El fitxer *rtsp_streams.launch* cumpleix la funció d'iniciar la transmisió de vídeo en temps real per RTSP.

Com l'arxiu anterior, consta de dues parts: càrrega del codi a executar i càrrega del codi de configuració. Es diferencia, però, en com crida el codi font. En lloc de cridar un node de ROS normal i corrent, s'executa un node *nodelet*, on posteriorment es carrega el codi a executar. Utilitzar *nodelets* dona la possibilitat de córrer múltiples algorismes dins el mateix procés, sense cost de computació a l'hora de comunicar-se entre processos.

```
1   <launch>
2
3     <!-- Start the RTSP server -->
4     <node pkg="nodelet" type="nodelet" name="standalone_nodelet" args="manager" output="screen"/>
5
6     <node pkg="nodelet" type="nodelet" name="Image2RTSPNodelet" args="load image2rtsp/Image2RTSPNodelet standalone_nodelet" output="screen">
7       <!-- Read the stream setup file -->
8       <rosparam command="load" file="$(find ros2ros)/config/stream_setup.yaml" />
9     </node>
10
11  </launch>
```

FIGURA 10.36: Codi de l'arxiu *rtsp_streams.launch*

Capítol 11

Conclusions

S'ha aconseguit crear un sistema capaç de comunicar nodes de ROS entre ells de manera personalitzada, alleugerant la càrrega de les comunicacions entre nodes de ROS de màquines separades. L'eina ROS2ROS funciona de forma genèrica, així que qualsevol projecte en pot fer ús, simplement modificant els arxius de configuració, i contempla tant la comunicació per topics com per serveis, permetent retransmetre topics on prima la latència i la velocitat mitjançant el protocol UDP, o el protocol que es desitgi.

S'ha estudiat la connexió dels mòdems òptics Luma, i s'ha obtingut retransmetre tots els missatges pensats originalment usant el paquet ROS2ROS: text, missatges estàndard de ROS, imatges i, el més complicat i que ha ocupat una part important del desenvolupament d'aquest projecte, vídeo.

El comportament dels Luma no ha complert amb les expectatives del fabricant, que anunciava 10 Mbits/s d'amplada de banda, però això ha jugat en favor del treball, ja que ha permès demostrar com el paquet desenvolupat funciona per a xarxes amb pèrdues i amb poca amplada de banda. A més a més, aquest fet ha incentivat a buscar i implementar millores, com l'enviament per RTSP per a la retransmissió de vídeo.

Es pot concloure que s'han assolit tots els objectius proposats al començament del projecte.

Capítol 12

Treball Futur

A l'hora de fer un treball individual, el temps i els recursos són factors molt limitadors, la falta dels quals va resultar en haver de descartar contingut d'aquest treball. A continuació es descriuen possibles millores que es podrien implementar en un futur.

12.1 Un únic YAML

Una de les millores que es pretenia fer pel paquet ROS2ROS, que es va descartar per acotar el termini d'entrega, era implementar l'ús d'un únic arxiu de configuració per a les dues màquines de la connexió. Els arxius *.yaml* que dicten el comportament del paquet a cada màquina són imatges pràcticament idèntiques entre els dispositius, canviant només l'especificació de si es vol rebre o enviar la informació. L'arxiu *config_sender.yaml* d'una màquina serà igual al *config_receiver.yaml* de l'altra, amb la diferència del contingut de la primera etiqueta.

Es podria incorporar una millora que permetés que en una sola màquina s'especifiquessin quins *topics* i serveis es volen compartir amb l'altre terminal, s'enviés aquesta informació a la màquina remota, i es creessin els paràmetres de configuració a partir de la configuració del primer dispositiu.

12.2 Millores de codi

El codi del paquet és funcional, però no és massa amigable amb l'usuari, ja que demana especificar molta informació per part d'aquest, quan aquesta ja existeix en el dispositiu. Una altra millora seria automatitzar la cerca d'aquestes dades, les quals ja es troben en la màquina, com ara el tipus de missatges que usen els *topics* i serveis, o l'adreça IP local. O, fins i tot, assignar de manera automàtica per quins ports es comuniquen els nodes de diferents *roscores*.

12.3 Creació instantània de subscriptors

Vist en l'apartat 8.5.3, existeixen *topics* de ROS que, quan un subscriptor s'hi connecta, el publicador envia paquets de capçalera necessaris per processar el contingut del tema abans de rebre res més.

Per la naturalesa del plantejament que s'ha fet en el paquet ROS2ROS, aquesta interacció no es porta a terme quan un subscriptor d'una màquina se subscriu a un

topic de l'altra, ja que aquest tema, en realitat, només consta d'un sol subscriptor. El que reenvia tot el que escolta del canal a l'altre *roscore*.

Per tant, una altra possible millora, descartada perquè caldria refer tot el funcionament del paquet, seria implementar una manera de detectar quan es crea un subscriptor a la màquina remota, i generar-ne un altre a la màquina local per cada subscriptor iniciat, el qual parli exclusivament amb el seu subscriptor designat. Així, quan s'engegués un subscriptor, es faria una nova connexió al *topic*, capturant els paquets de capçalera de nou.

Apèndix A

ros2ros

ROS package to comunicate between two separate rosmasters

A.1 Description

This package allows two separate ROS machines with separate rosmasters to lisent and publish to eachother's topics and proxy eachother's services.

A.2 Dependencies

- ROS
- CV2 (OpenCV)
- gstreamer libs:

```
1 sudo apt-get install libgstreamer-plugins-base1.0-dev libgstreamer-
    plugins-good1.0-dev libgstreamer-plugins-bad1.0-dev
    libgstrtspserver-1.0-dev gstreamer1.0-plugins-ugly gstreamer1.0-
    plugins-bad
```

A.3 Usage

For each roscore, use the following command to enable the publisher/listener comunication between rosmasters.

`roslaunch ros2ros comunicacio_topics.launch`

To enable the service's comunication, use the following command

`roslaunch ros2ros comunicacio_services.launch`

A.4 YAML Setup

In order to work properly, each machine must have the config YAMLs setup to allow comunication from the other machine.

A.4.1 config_receiver.yaml

This YAML configures which topics are read from external connections.

The local address must be given under the label "address". Decleare each topic to be recievd under the label "topics". There can be declared as many topics as desired, as long as they are given a unique port. Each topic must provide the topic's name, the kind of messages it uses, and the port it is expected to be read from. It can additionally provide which protocol is used to send the topics information (tcp, udp or rtsp). By default it will be set as UDP. If the protocol set is RTSP, it must be provided the url of the stream. Currenty, only video-based streams are supported. Everything must be under the label "receiver".

The file should follow this pattern:

```
receiver:
  address: 'xxx.xxx.xxx.xxx' # Local address

  topics:
    example_topic:
      topic: "/topic_name"
      type: "package_family.msg.packageName"
      port: 10006
      protocol: "tcp"
    example_stream:
      topic: "/topic_name"
      type: "sensor_msgs.msg.Image"
      port: 10010
      protocol: "rtsp"
      rtsp_url: "rtsp://ipaddressofserver:port/mountpoint"
```

Each declared topic should be also delcared with the same parameters in the file config*sender.yaml of the opposite machine, except for RTSP streams. Those must be declared in the file stream*setup.yaml.

A.4.2 config_sender.yaml

This YAML configures which topics are sent to external connections.

The target connection address must be given under the label "address". Decleare each topic to be sent under the label "topics". There can be declared as many topics as desired, as long as they are given a unique port. Each topic must provide the topic's name, the kind of messages it uses, and the port it is expected to be read from. It can additionally provide which protocol is used to send the topics information (tcp or udp). By default it will be set as UDP. To send data through RTSP, refer to the stream_setup.yaml file. Everything must be under the label "receiver".

The file should follow this pattern:

```
receiver:
  address: 'xxx.xxx.xxx.xxx' # Target address

  topics:
    example_topic:
      topic: "/topic_name"
      type: "package_family.msg.packageName"
```

```
      port: 10006
      protocol: "tcp"
```

Each declared topic should be also delcared with the same parameters in the file config_receiver.yaml of the opposite machine.

A.4.3 stream_setup.yaml

This YAML configures which Image based topics are sent to external connections via RTSP.

The port used to send the streams must be given under the label "port". Decleare each stream to be recievd under the label "streams". There can be declared as many streams as desired, as long as they are given a unique mountpoint. Each stream must provide the source name, the source type, the mountpoint of the stream, the video restrictions, and the target bitrate.

The file should follow this pattern:

```
port: "8554"
streams:
  example_stream:
    type: topic
    source: /videofile/image_raw # Image rostopic. It must be raw image!
                                 # Don't use transport topics!
    mountpoint: /back

    # Constraints of the video stream
    caps: video/x-raw,framerate=10/1,width=640,height=480
    bitrate: 500 # Target bitrate of the stream
```

Each declared stream should be also delcared with the same parameters in the file config_receiver.yaml of the opposite machine.

A.4.4 config*tcp*service.yaml

This YAML configures which services are comunicated with external sources.

The local address must be given under the label "address". Decleare each service under the label "services". There can be declared as many services as desired, as long as they are given a unique port. Each service must provide the service's name, the kind of service it is, and the port it is expected to be read from. Everything must be under the label "service_server".

The file should follow this pattern:

```
service_server:
  address: 'xxx.xxx.xxx.xxx' # Local address

  services:
    example_topic:
      service: "/service_name"
```

```
      type: "service_family.srv.serviceClassName"
      port: 10006
```

Each declared service should be also delcared with the same parameters in the file config*tcp*client.yaml of the opposite machine.

A.4.5 config*tcp*client.yaml

This YAML configures which external services are able to be proxied.

The target connection address must be given under the label "address". Decleare each service under the label "services". There can be declared as many services as desired, as long as they are given a unique port. Each service must provide the service's name, the kind of service it is, and the port it is expected to be read from. Everything must be under the label "service_client".

The file should follow this pattern:

```
service_client:
  address: 'xxx.xxx.xxx.xxx' # Target address

  services:
    example_topic:
      service: "/service_name"
      type: "service_family.srv.serviceClassName"
      port: 10006
```

Each declared service should be also delcared with the same parameters in the file config*tcp*service.yaml of the opposite machine.

Bibliografia

[1] *Portal de treball Talent.com*. URL: https://es.talent.com/.
[2] *Portal de la empressa Hydromea*. URL: https://www.hydromea.com/.
[3] CircusMonkey. *ROS_RTSP*. URL: https://github.com/CircusMonkey/ros_rtsp.
[4] awesomebytes. *Portal de treball Talent.com*. URL: http://wiki.ros.org/video_stream_opencv.
[5] Enric Pagès Agustí. *Paquet ROS2ROS*. URL: https://github.com/pagesenric/ros2ros.
[6] hermanoid. *Forum de ROS d'on s'ha tert la funció* $CV2_TO_IMGMSG$. URL: https://answers.ros.org/question/350904/cv_bridge-throws-boost-import-error-in-python-3-and-ros-melodic/.