73988

TOSCO, SEBASTIAN JOE
Arquitectura de labo

2014 73988

ARQUITECTURA DE LABORATORIOS REMOTOS PARA LA ENSEÑANZA DE LA INGENIERÍA

por

Sebastián Joel Tosco

Tesis presentada para la obtención del grado de Magíster en Ciencias de la Ingeniería
Mención: Ingeniería Eléctrica

FACULTAD DE INGENIERÍA
UNIVERSIDAD NACIONAL DE RÍO CUARTO
Año 2014

MFN:
Clasif:
T.955

ARQUITECTURA DE LABORATORIOS REMOTOS PARA LA ENSEÑANZA DE LA INGENIERÍA

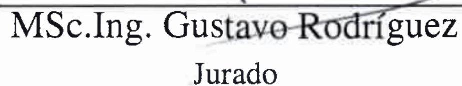
por

Sebastián J. Tosco

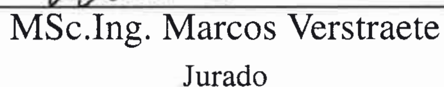
MSc.Ing. Fernando Corteggiano
Director

APROBADA POR:

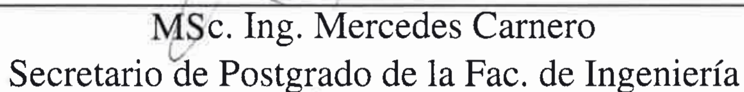
MSc.Ing. Aldo Crespo
Jurado



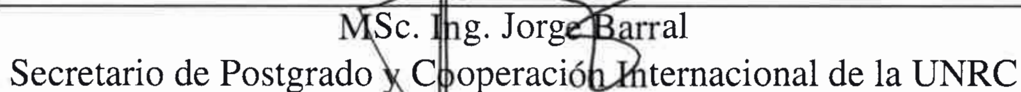
MSc.Ing. Gustavo Rodríguez
Jurado



MSc.Ing. Marcos Verstraete
Jurado



MSc. Ing. Mercedes Carnero
Secretario de Postgrado de la Fac. de Ingeniería



MSc. Ing. Jorge Barral
Secretario de Postgrado y Cooperación Internacional de la UNRC

Año 2014

Dedicado a mi amada esposa, Noelia, quién me ha dado su apoyo, comprensión y ánimo, incluso en los momentos cuando más lo he necesidado. También a mis padres Danilo y María del Carmen por su tierno cariño desde siempre...

Agradecimientos

Esta obra es el resultado de la inversión diligente de mucho tiempo y esfuerzo que estoy muy dichoso de haber podido realizar. Sin embargo, si bien la autoría es de carácter unipersonal, eso no significa que el autor haya de ser el único que se deba llevar todo el crédito...
Par empezar, quiero agradecer profundamente a mi director Fernando Corteggiano, quién me acompañó en la realización de esta tesis y durante las diversas tareas e implementaciones que se han realizado durante mi carrera de postgrado. Realmente, su apoyo lo define más que sólo como un director, más bien: *un gran compañero...*
También, cabe aclarar que ha sido muy animador recibir el apoyo de mis colegas que siempre aportaron las palabras de aliento necesarios para motivar la superación de los obstáculos que han surgido mientras se recorría esta etapa de capacitación profesional.
Además, es digno de mencionar que fueron buenos disparadores las ideas de varios compañeros del Depto. de Química de la Facultad de Ingeniería (UNRC) y por ello estoy especialmente agradecido.
Finalmente, siempre agradecido al apoyo de mi esposa (querida compañera de vida) y a mis padres, es que presento esta tesis de postgrado... un compendio de conocimiento e investigación y, entre líneas, un testimonio de dedicación, esfuerzo y valor.

Índice general

Agradecimientos V

Índice General VI

Índice de Cuadros XI

Índice de Figuras XII

Resumen XV

Abstract XVII

I. INTRODUCCIÓN 1

1.1. Descripción del problema 2

1.2. Objetivos 3

1.2.1. Objetivo general 3

1.2.2. Objetivos particulares 3

1.3. Organización de la tesis 3

II. LOS LABORATORIOS REMOTOS Y LA TELEOPERACIÓN 5

2.1. Los laboratorios remotos: el ayer y el hoy 5

2.2. La teleoperación: definición y aplicaciones 7

2.2.1. Los sistemas de control y la teleoperación 8

2.3. La importancia de los laboratorios remotos en la enseñanza de la ingeniería 9

2.4. Conclusiones 12

III. TECNOLOGÍAS PARA EL DISEÑO DE LABORATORIOS REMOTOS 13

3.1. Introducción 13

3.2. Middleware: Taxonomía y Definiciones 13

3.3. Capa Intermedia orientada a Mensajes (MOM) 16

3.3.1. ¿Qué implica que un sistema sea asincrónico? 17

3.3.2. Ventajas de Middlewares orientados a mensajes 18

3.4. Comparación entre MOMs y RPC 18

3.4.1. Introducción al Procedimiento de Llamada Remota tradicional (RPC) 19
3.4.1.1. Algunos detalles sobre RPC 20
3.4.2. Introducción a MOM 20
3.4.2.1. Algunos detalles sobre MOM 21
3.4.3. Cuándo usar MOM y cuando RPC 22
3.5. Modelo de intercambio de mensajes en los MOM 23
3.5.1. Nombres jerárquicos de tópicos 23
3.5.2. Comparación entre ambos modelos de mensajeo 24
3.6. Estructura de Mensajes 24
3.7. Servicios comunes de los MOMs 25
3.7.1. Filtrado de mensajes 25
3.7.2. Transacciones 25
3.7.3. Garantías de entrega de mensajes 26
3.7.4. Formatos de mensajes 27
3.7.5. Balance de carga 27
3.7.6. Agrupamiento (Clustering) 27
3.8. Desempeño de los sistemas basados en mensajes 27
3.9. Conclusiones 28

IV. LA CAPA INTERMEDIA: ROBOTIC OPERATION SYSTEM (ROS) 31
4.1. Aspectos generales 31
4.2. Arquitectura del sistema ROS 32
4.3. Conceptos de la arquitectura ROS 33
4.3.1. Nivel de archivos de sistema 34
4.3.2. Nivel de funcionamiento 34
4.3.3. Nivel de comunidad 36
4.4. Comparación de ROS con otros Middleware para robótica 38
4.5. Rosbridge: Un "puente" entre el ROS y la Web 39
4.6. MJPG-streamer: Útil herramienta para la visualización de streaming de video 42
4.7. Conclusiones 44

V. USO DE SISTEMAS EMBEBIDOS COMO NODOS DE ADQUISICIÓN Y PROCESAMIENTO 45
5.1. Caracterización de los sistemas embebidos 45
5.2. Arquitectura básica 46
5.3. Aplicaciones de los sistemas embebidos 46
5.4. Beaglebone: un microcomputador de buenas prestaciones 47
5.4.1. Bancos de expansión 50

5.4.2. Primera conexión con BeagleBone 51
5.5. Uso de la BeagleBone como nodo ROS 51
5.5.1. Primeros pasos para crear un paquete donde correrá el nodo . . . 52
5.5.2. Un nodo ejemplo 53
5.5.3. Un nodo servidor MJPEG 55
5.6. Conclusiones 57

VI. IMPLEMENTACIÓN DE UN CASO DE ESTUDIO PARA LA ARQUITECTURA PROPUESTA 59
6.1. Escenario de prueba 59
6.2. Nodos ROS del caso de estudio 62
6.2.1. Nodo de video 63
6.2.2. Nodo planta 64
6.2.3. Nodo PID 65
6.2.4. Nodo red 66
6.2.5. Nodo de procesamiento de imágenes 66
6.3. Resultados 66
6.3.1. Interfaz gráfica html 66
6.3.2. Tráfico de datos en el sistema 68
6.3.3. Influencia de los parámetros de control 68
6.3.4. Influencia de los retardos 69
6.3.4.1. Escenario 1 71
6.3.4.2. Escenario 2 72
6.3.4.3. Escenario 3 72
6.3.4.4. Escenario 4 72
6.3.5. Conclusiones 73

VIICONCLUSIONES Y TRABAJO A FUTURO 75
7.1. Introducción 75
7.2. Trabajo a futuro 76

Bibliografía 79

Anexos 83
Anexo A 85
Anexo B 93
Anexo C 101
Anexo D 105
Anexo E 107
Anexo F 111

Anexo G 117
Anexo H 119

Índice de cuadros

3.2.1. Taxonomía de Middlewares [16] 14
3.4.1. Resumen de algunos conceptos 19
3.7.1. Características de una transacción 25

4.4.1. Comparativa entre algunos frameworks para robótica 38

5.2.1. Partes fundamentales de un ordenador empotrado 46

6.2.1. Archivo rc_sim.launch 63
6.3.1. Estadísticas de red WLAN (wifi) 70
6.3.2. Estadísticas de la comunicación ROS_MASTER - Planta (sin wifi) 70

Índice de figuras

2.2.1. Arquitectura general de un sistema teleoperado 7
2.3.1. Servicios de la web 2.0 [13] 10

3.2.1. Middleware y sistemas distribuidos ([18]) 14
3.3.1. Diferencia entre interacción sincrónica y asincrónica [23] 17
3.4.1. Esquema de funcionamiento de RPC [14] 20
3.4.2. Esquema de sistema MOM [14] 21
3.5.1. Modelos de intercambio de mensajes [14] 23
3.7.1. Roles en una transacción [14] 26

4.1.1. Vista del sitio oficial de ROS 32
4.2.1. Arquitectura general sistema ROS [31] 33
4.3.1. Arquitectura general sistema ROS [32] 36
4.3.2. Interacción entre Master y nodos.[33][34] 37
4.4.1. Ejemplo del "core" de ROS lanzado en una PC llamada "Groovy" 40
4.5.1. Ejemplo de Rosbridge lanzado en una PC llamada "Groovy" y esperando conexiones 42
4.6.1. Captura de streaming de video generado desde una webcam y transmitido mediante MJPG-Streamer 43

5.4.1. Vista de un beaglebone 48
5.4.2. Diagrama en bloques del AM3359 49
5.4.3. Tabla de consumo del Beaglebone 49
5.4.4. Asignaciones de expansión P8 (Beaglebone) 50
5.4.5. Asignaciones de expansión P9 (Beaglebone) 51
5.5.1. Ejemplo de carpeta de un paquete llamado rc_sim creado en un BB 53
5.5.2. Contenido del manifiesto de un paquete llamado rc_sim creado en un BB 53
5.5.3. Sección del código de nodo_planta.py 54
5.5.4. Sección del código de nodo_camara.sh 55
5.5.5. Video capturado por el nodo_camara corriendo sobre el BB 56
5.5.6. Contenido del archivo "pru.launch" del nodo "nodo_cam" 56
5.5.7. Salida en terminal al poner en marcha el nodo ROS "nodo_camara" . . . 56

6.1.1. Caso de estudio . . . 60
6.1.2. "Servidor principal" distribuido en tres dispositivos . . . 61
6.1.3. Manifiesto del paquete "rc_sim" . . . 62
6.2.1. Archivo nodo_camara.sh . . . 64
6.2.2. Tráfico de video desde el nodo_cam . . . 64
6.2.3. Circuito RC . . . 65
6.2.4. Tasa de publicación de tópicos Nivel(/Y) y Control(/U) . . . 65
6.2.5. Nodo de procesamiento en funcionamiento (entorno web versión *alfa*) . . 67
6.3.1. Visualización de la interfaz web en 2 dispositivos distintos . . . 67
6.3.2. Tráfico medido desde el cliente . . . 68
6.3.3. Histograma y respuesta del sistema (Nivel) bajo dos configuraciones distintas . . . 69
6.3.4. Comando de interfaz web para manejo del nodo_red . . . 70
6.3.5. Respuesta del sistema (Nivel) a una perturbación para dos configuraciones distintas (escenario 1) . . . 71
6.3.6. Tiempo de subida (escenario 1) . . . 71
6.3.7. Respuesta del sistema (Nivel) a una perturbación e histograma (escenario 2) 72
6.3.8. Respuesta del sistema (Nivel) a una perturbación e histograma (escenario 3) 73
6.3.9. Respuesta del sistema (Nivel) a una perturbación e histograma (escenario 4) 73

UNIVERSIDAD NACIONAL DE RIO CUARTO
BIBLIOTECA

Resumen

Actualmente, el uso de laboratorios remotos a través de Internet permite complementar la enseñanza de la ingeniería, especialmente en lo que a prácticas experimentales se refiere. En dicho tipo de estructura, se utilizan las redes de telecomunicaciones para controlar a distancia los procesos bajo estudio, a la vez que una cámara de video sirve para visualizar el funcionamiento del sistema. Una ventaja muy marcada de este tipo de prácticas es que brindan la posibilidad de teleoperar un sistema con buena precisión, salvando las distancias y, de ser necesario, evitando riesgos para la salud e incluso la vida.

En esta tesis se propone una arquitectura de laboratorio remoto instrumentada sobre una red de telecomunicaciones y accesible al usuario a través de una interfaz web. Para ello, tras una introducción breve al concepto de laboratorios remotos, se estudian algunas de las tecnologías que se utilizan para comunicar los sistemas distribuidos y se explica su adaptación a los laboratorios remotos.

Posteriormente se analiza un meta-sistema operativo de código abierto pensado para robótica que tiene la capacidad de obtener, construir, escribir y ejecutar código en *varios equipos*. El análisis se centrará en la posibilidad de usarlo para transmitir y procesar variables entre diversos dispositivos y sensores interconectados mediante una red de telecomunicaciones.

Finalmente se exponen los resultados obtenidos sobre un caso de estudio particular: un laboratorio remoto que medirá variables y comunicará dicha información al usuario final quién puede controlar manualmente dicho nivel a través de la red.

Las pruebas realizadas demuestran que si bien se presenta cierto retraso en la comunicación, esta es tolerable para la aplicación. Además, utilizando las herramientas apropiadas se puede hacer un buen uso del ancho de banda disponible en la red, a la vez que la interfaz para el alumno resulta propicia para el aprendizaje.

Abstract

Currently, the use of remote labs via the Internet can complement the engineering education, especially as regards experimental practices. In this type of structure, telecommunications networks are used to remotely control the processes being studied, while a video camera is used to visualize the operation of the system. A marked advantage of this type of practice is to provide the ability to teleoperate a system with good accuracy, bridging the gap and, if necessary, avoiding risks to health and even life.

This thesis proposes a remote laboratory architecture instrumented on a telecommunications network and accessible to the user through a web interface. For this, after a brief introduction to the concept of remote laboratories, explores some of the technologies that are used to communicate distributed systems and explains their adaptation to remote laboratories.

Subsequently, analizes a open source meta-operating system designed for robot that has the ability to obtain, build, write and execute code on multiple computers. The analysis will focus on the possibility of using it to transmit and process variables between various devices and sensors interconnected by a telecommunications network.

Finally, this text presents the results obtained on a particular case of study: remote laboratory that measures variables and communicates that information to the end user, who may control this level manually through the network.

Tests have shown that although some delay is presented in the media, this is tolerable for the application. Moreover, using the proper tools can make good use of the available bandwidth in the network, while the student's interface is conducive to learning .

CAPÍTULO I   INTRODUCCIÓN

Con el avance de la tecnología en el campo de la computación y las redes, el estudio de la interacción entre las comunicaciones y los sistemas de control se ha vuelto atractivo para la investigación y el desarrollo. Pero esto no es todo. En el ámbito educativo, también existe mucho interés en dicha integración, especialmente en las áreas afines a la enseñanza de las ciencias y la ingeniería.

Uno de los problemas más frecuentes en el ámbito educativo son los costes de implementación y mantenimiento de laboratorios apropiados y eficaces para el alumnado. Por tanto, entre los proyectos que se están llevando a cabo para la actualización y mejora de la enseñanza se cuentan los que tienen que ver con los Laboratorios Remotos (LR). Sobran los motívos por los cuales se está optando por implementar sistemas que permitan compartir experiencias de laboratorio a distancia, de tal manera que el alumno pueda "sentir" que maneja y controla el proceso bajo estudio "como si estuviera presente en el lugar", lo cual le permite adquirir habilidades en el manejo de dispositivos e instrumental real sin importar su ubicación geográfica. Dicha interacción, sumamente necesaria para el proceso de aprendizaje, ha llevado a incluir el concepto de teleoperación en el diseño de los LR así como también requerimientos de transparencia, facilidad de operación, fiabilidad en el sistema de control y costos.

En dicho proceso de diseño, salen a relucir las numerosas tecnologías y estándares que pueden coexistir en los equipos de un laboratorio experimental y que, a la hora de establecer vinculaciones entre ellos, representan un desafío. Además, la teleoperación implica la necesidad de vincular sensores y actuadores para reportar estados de situación y generar respuestas apropiadas. Dicho concepto está presente en el análisis y requiere que la arquitectura que se proponga para el LR tenga características determinadas de comportamiento ante una red de telecomunicaciones (por ejemplo una red local, o incluso Internet).

En este sentido actualmente hay un gran auge de las Capas Intermedias de Software Orientadas a Mensajes (MOM, por sus siglas en inglés). Dichas arquitecturas permiten independizar al usuario de los procesos a bajo nivel y concentrarse en la implementación a alto nivel, con confiabilidad y eficiencia. Dentro de el amplio abanico de MOMs se puede encontrar un grupo que resulta apropiado para robótica y por tanto muy afín a las necesidades planteadas de comunicación entre elementos distribuidos y teleoperación.

Ahora bien, habiendo encontrado una tecnología que resulte apropiada para el diseño e implementación de LR para enseñanza de ciencias e ingeniería, el interrogante supremo termina siendo cuál elegir... Lo cierto es que más allá de gustos, se pueden pensar en algunos criterios objetivos que permitan una elección a conciencia. En este particular, se presenta como buena opción el Sistema Operativo Robótico (ROS, por sus siglas en inglés). Dicho metasistema operativo presenta muy buenas características que son apreciadas en el diseño de LR, entre estas: buen conjunto de librerías de trabajo, curva de aprendizaje moderada, buena documentación, multilenguaje, código libre, entre otras.

Con este esquema de situación (necesidad de LR, sistemas MOM para robótica y la elección de ROS), es que se puede concebir un prototipo que muestre el funcionamiento de una arquitectura de comunicación entre sistemas distribuidos para teleoperar un LR dedicado a la enseñanza de las ciencias y la ingeniería. En este marco, se trabajó y finalmente se pudo crear un prototipo de LR que emula la carga de un tanque mediante un RC. Es interesante destacar que si bien este laboratorio tuvo como interés inicial la anteriormente mencionada temática, también resulta interesante para otras cosas, por ejemplo: laboratorio de circuitos ó análisis de redes IP.

1.1. Descripción del problema

Como se comentó anteriormente, dentro de la enseñanza de las ciencias y la ingeniería, los laboratorio experimentales representan una herramienta de enseñanza fundamental. Sin embargo, dado que el concepto tradicional de laboratorio de prácticas, en el marco de las enseñanzas de tipo científico o tecnológico, presenta en la actualidad una serie de dificultades (tanto económicas como de infraestructura), hoy en día se han empezado a desarrollar laboratorios teleoperados a través de redes de telecomunicaciones.

Se puede definir un *laboratorio teleoperado* como el *conjunto de equipos e instrumentos reales remotamente controlados, utilizando una interfaz específica.* Esta idea no es nueva, dado que se viene utilizando desde hace décadas en la industria. Sin embargo, el advenimiento de nuevas tecnologías en lo que a redes y computadores se refiere genera una constante actualización en las estructuras y configuración de los equipos.

Por otro lado, la aplicación de técnicas matemáticas al análisis de datos e imágenes ha permitido que este campo de la investigación tenga alta aplicabilidad, tanto en el ambiente académico como en la industria. De hecho, estudios recientes sobre la posibilidad de medir temperaturas mediante una cámara de video incluso en circunstancias donde la utilización de un sensor es inviable, abre las puertas del pensamiento al abanico de posibilidades que presenta el procesamiento de imágenes. También, el análisis de datos provenientes de multiples sensores y la toma de decisiones para realizar un control óptimo sobre un sistema son hoy en día temáticas muy bien estudiadas.

Por último, la evolución de las redes de telecomunicaciones permite la implementación de transmisión de datos en tiempo real. En la actualidad es posible disponer de ancho de banda razonables a precios módicos, lo que permite el acceso remoto a equipos e instalaciones de un laboratorio real.

En el marco descrito entonces, el problema se puede formular así: ¿Es posible proponer una arquitectura para LR orientado a la enseñanza de las ciencias y la ingeniería de tal manera que resulte eficiente incluso cuando los elementos del laboratorio están distribuidos en el espacio y las comunicaciones (entre dispositivos y con respecto al usuario) aparece mediada por una red de telecomunicaciones (ya sea local o Internet)? Evidentemente, contestar este interrogante requiere un estudio de tecnologías y arquitecturas plausibles de utilizar en laboratorios remotos.

1.2. Objetivos

El desarrollo del presente trabajo persigue un objetivo principal y varios objetivos particulares.

1.2.1. Objetivo general

Proponer, analizar y aplicar a un caso particular una arquitectura de laboratorio remoto para enseñanza de las ciencias y la ingeniería.

1.2.2. Objetivos particulares

En pro de alcanzar el macro-objetivo descripto anteriormente, se plantean otros objetivos secundarios:

- Estudio de arquitecturas relacionadas con LR.
- Análisis del comportamiento de los sistemas teleoperados en precencia de una red de comunicaciones.
- Escojer una arquitectura y determinar su utilidad en el diseño de un LR.
- Implementación de un prototipo que ilustre la arquitectura escogida.

1.3. Organización de la tesis

El presente escrito plasma la información con un órden lógico de razonamiento y de proceso, yendo desde lo general hasta lo específico. Así, tras este primer capítulo intro-

ductorio, se puede hallar un segundo capítulo que comenta el porqué de la existencia de los LR y su relación con el concepto de teleoperación.

Posteriormente, en el capítulo 3, se muestran conceptos y descripciones relacionados con las tecnologías que son plausibles de utilizar al diseñar laboratorios remotos. De todas estas, se ha escogido una, el Robot Operation System (ROS), el cual se explica en el cuarto capítulo.

En el capítulo 5 se muestra las posibilidades que la arquitectura elegida presenta mediante la utilización de hardware embebido de bajo costo plausible de utilizar en los LR. Se analiza un hardware en particular dentro de esta gama: el BeagleBone.

Hacia el final del documento, en el sexto capítulo se presenta y analiza un prototipo de laboratorio remoto que emula la carga de un tanque de agua mediante un sistema RC de primer orden. También, se plasman resultados del comportamiento del sistema ante cambios de las condiciones de la red de comunicaciones y se visualizan efectos cuantitativos y cualitativos en el comportamiento del sistema y el control del mismo.

Finalmente, en el séptimo capítulo, se muestran las conclusiones generales a las que se arrivaron al realizar el presente trabajo y se deja constancia de trabajos futuros que pueden proyectarse a partir de esta tesis de postgrado.

CAPÍTULO II LOS LABORATORIOS REMOTOS Y LA TELEOPERACIÓN

La evolución de las redes de telecomunicaciones ha permitido su utilización en múltiples proyectos de ingeniería. Entre dichos proyectos se cuentan los que tienen que ver con la educación, por ejemplo los Laboratorios Remotos (LR). Por diversos motívos, económicos y de infraestructura entre otros, se está optando por implementar sistemas que permitan compartir experiencias de laboratorio a distancia, de tal manera que el alumno pueda "sentir" que maneja y controla el proceso bajo estudio "como si estuviera presente en el lugar", lo cual le permite adquirir habilidades en el manejo de dispositivos e instrumental real sin importar su ubicación geográfica [1].

En el presente capítulo se analiza el estado del arte de los laboratorios remotos y el papel que desempeña la teleoperación en el funcionamiento de los mismos.

2.1. Los laboratorios remotos: el ayer y el hoy

Desde antaño la experimientación ha sido un pilar en la enseñanza de las ciencias. Sin embargo, el concepto tradicional de los laboratorios de prácticas, en el marco de las enseñanzas de tipo científico o tecnológico, plantea en la actualidad una serie de dificultades, especialmente de tres tipos:

- El elevado coste, tanto de adquisición como de mantenimiento, de los equipos necesarios.
- La casi imposibilidad de disponer en los centros educativos de salas de laboratorio con los recursos suficientes para la totalidad de los alumnos que cursan materias de este tipo.
- La no disponibilidad de los laboratorios en cualquier horario, dado que su uso presencial requiere, en general, la presencia del profesor. Esto lleva a una infrautilización de los recursos disponibles.

En virtud a estas problemáticas hoy en día se han comenzado a desarrollar laboratorios teleoperados o remotos a través de redes de telecomunicaciones.

Una experiencia de uso de un Laboratorio Remoto puede ser vista como un usuario con

un ordenador en un lugar distante que controla remotamente un experimento en una localización específica. Dispositivos de estas características vienen siendo utilizados desde hace más de 20 años en la industria, por ejemplo en el campo aeroespacial para el control de naves y telescopios espaciales. Las primeras aplicaciones de telemanipulación a través de Internet tuvieron lugar hace poco más de una década principalmente en el campo de la robótica. En España, una Universidad pionera fue La Universidad de Valladolid, la cual comenzó a trabajar en este campo 1998. Sin embargo se puede afirmar que la migración de este tipo de aplicaciones al ámbito de la educación, ha sido posterior y ha venido ligada por una parte al aumento del ancho de banda disponible para el acceso a Internet por parte de los usuarios, y al abaratamiento tanto de las tarifas de conexión de alta velocidad como al de los propios ordenadores[2].
Si bien no existen reglas axiomáticas se puede, en virtud de la experiencia, establecer algunas pautas de diseño básicas a la hora de implementar un laboratorio remoto[2]:

- Los actuadores (ej.: motores) y sensores (ej.: termocuplas) deben poder ser accionados y leídos por un usuario remoto que se conecta al laboratorio a través de la Web (Internet).
- Debe existir una camara que posibilite una vista general de la situación para que el operador remoto sienta que tiene el control del experimento, requisito fundamental para alcanzar una motivación óptima.
- El sistema debe contar con facilidad de acceso y sencillez de manejo.
- Las medidas tienen que ser rápidas y deben poderse visualizar gráficamente los resultados para que puedan ser validados y en caso contrario repetir la toma de datos.
- Además el usuario debe encontrar toda la información necesaria (manual de funcionamiento, guión de la práctica, etc.) en la misma aplicación.
- Por último el usuario ha de tener también la posibilidad de realizar consultas y presentar una memoria del experimento para que pueda ser corregida y convenientemente evaluada.

Según algunos autores [3], puede hablarse de dos generaciones de laboratorios remotos. La primera es aquella en la cual el lazo de control del proceso se establece del lado del servidor, es decir en planta, mientras del lado del usuario solo se establecen los parámetros y se hace el monitoreo de la situación.

Una segunda generación, inspirado en el mundo industrial, aspira a transformar los laboratorios controlados remotamente en sistemas controlados a través de la red (Networked Control Systems, NCS). En estos sistemas el control está del lado del cliente y el usuario puede cambiar la estructura de control en vez de solo unos cuantos parámetros. Además,

hay solo un lazo de control y este se cierra a través de la red.
A continuación se dan más detalles sobre este tipo de sistema.

2.2. La teleoperación: definición y aplicaciones

Se entiende por "teleoperación" a la extensión de la percepción de una persona, la decisión de hacer, y la capacidad de manipulación a distancia[4]. Los sistemas de teleoperación de hoy en día permiten la interacción con los ambientes a distancia y también se puede escalar la fuerza humana y el movimiento para conseguir capacidades de acción pese a restricciones de tamaño, fuerza, accesibilidad o incluso peligro para la vida. Las aplicaciones clásicas van desde la manipulación de residuos tóxicos/nucleares/inflamables hasta aplicaciones de rescate en entornos variados y cirugías de mínima invasión, pasando por la nanorobótica y aplicaciones orientadas a la educación y el entretenimiento[4].

En un sistema teleoperado, los comandos de control y las mediciones realizadas por sensores se transmiten utilizando diversos medios, entre ellos, microondas, radiofrecuencias y redes de computadoras. Dada la evolución de los sistemas de comunicaciones, los programas relacionados y los lenguajes de programación, hoy en día existen soluciones simples y económicas para el desarrollo de sistemas operados a distancia que utilicen Internet como medio de comunicación de alcance mundial [5].

Si bien se han desarrollado varios sistemas teleoperados sobre Internet (por ejemplo [6, 7, 8]), cada uno con sus particularidades, en general hay ciertas características que están presentes inherentemente en todos ellos y que nos dan una visión general del funcionamiento de un esquema de operación remoto; éstas pueden verse en la figura 2.2.1.

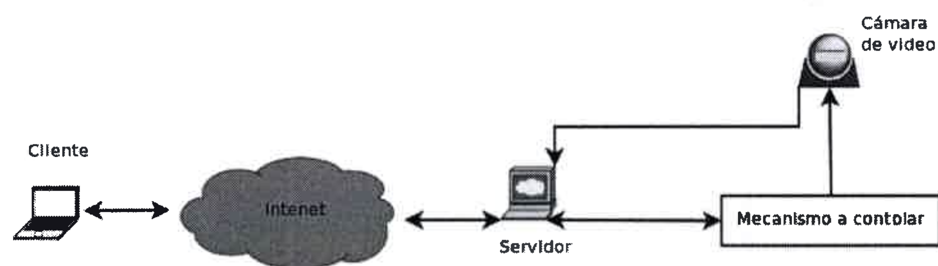


Figura 2.2.1.: Arquitectura general de un sistema teleoperado

En la figura se observa a un usuario con un ordenador cliente que controla a través de Internet un experimento real en una localización remota. El experimento se realiza mediante diferentes mecanismos, motores y sensores que son controlados por un servidor Web y un interfaz gráfico. Una cámara permite seguir en tiempo real toda la experiencia [2].

Como se puede apreciar, el sistema consta de dos grandes partes: cliente y servidor [5]. El primero se consuma en la computadora del usuario, donde él puede monitorear todo

el sistema y realizar las acciones a través de una interfaz gráfica de usuario (usualmente denominada GUI). Dicha interfaz debe poder manejar la comunicación bilateral que se establece entre el cliente y el sistema (red + sistema remoto), esto incluye presentar clara y convenientemente los datos recibidos, así como también enviar a la parte del servidor los comandos de control que elige el usuario. Dicha interfaz se prefiere amigable, con opciones de ayuda e íconos apropiados. También, existen nuevos enfoques donde incluso la interactividad y el video se mezclan en una composición muy novedosa y atrayente[9].

La parte del servidor, generalmente ubicada en el mismo sitio donde está el mecanismo o proceso a controlar, se ocupa de las políticas de acceso, servidor web y control local. Además, este servidor también recibe el flujo de video de la cámara el cual contiene la información visual sobre el estado de situación del mecanismo bajo estudio y la retransmite a los clientes vía web.

2.2.1. Los sistemas de control y la teleoperación

Cuando se concibió la teleoperación, se vislumbró la posibilidad de realizar tareas peligrosas en entornos remotos manteniendo al operador a una distancia segura. Sin embargo, los canales de comunicaciones genéricos, por ejemplo Internet, generan desafíos para cumplimentar dicho objetivo. Entre estos problemas, el tiempo de retardo se identifica como el principal factor limitante para el logro de un desempeño satisfactorio. Además, generalmente se requiere un ancho de banda considerable y un equilibrio apropiado entre el tiempo de muestreo adoptado y la tasa de envío de datos. Desde el punto de vista de la teoría de control, estos problemas se vuelven más desafiantes cuando la planta controlada remotamente presenta dinámicas rápidas e inestables, de tal manera que los tiempos de accionamiento se reducen considerablemente. ¿Qué mecanismo o modelo de control utilizar? Las pruebas realizadas demuestran que la estrategia que mejor se adapta a los retrasos en la red es la utilización del Modelo de Control Predictivo (MPC, por sus siglas en inglés)[3].

Sin embargo, cuando se requiere un intercambio rápido de datos, la aplicación de MPC, como tal, no es funcional debido al tiempo necesario para llevar a cabo en línea la optimización requerida por el controlador predictivo. Un enfoque diferente es el llamado Control Anticipativo, el cual toma algunas ideas del MPC y soluciona la dificultad mencionada. Se estima el estado de la planta sobre la base de un modelo teniendo en cuenta los retrasos pero sin realizar optimización. La secuencia de control resultante puede no ser óptima pero su cálculo tiene una ranura de tiempo insignificante en comparación con el MPC.

Es digno de mención que la demanda de sistemas de control en red (NCS) no sólo vienen desde el mundo industrializado. De hecho la experimentación remota a través de inter-

net es un ámbito de investigación creciente y actualmente de desarrollo. Una nueva línea de investigación, cuyo origen se relaciona con el mundo industrial, persigue la transformación de los laboratorios remotamente controlados en NCS. En este nuevo enfoque, el controlador está en el lado del cliente, y el usuario en realidad puede modificar su estructura en lugar de unos pocos parámetros. Sólo hay un bucle y se cierra a través de la red.

Los NCS tienen que cumplir con algunos requisitos:

- El diseño de aplicaciones que proporcionen la interacción con otros servicios o aplicaciones.
- Ser independientes de los servicios de red.
- Ser fiables y estar siempre disponibles (confiabilidad alta).

El software que posibilita los requisitos antes mencionados se denomina *capa intermedia* (middleware). Gracias a éste, los diseñadores pueden concentrarse en el control del sistema e independizarse de los detalles de la red subyacente y sus peculiaridades.

Desde un punto de vista educativo, cuando un alumno opera un NCS para experimentación remota no verá diferencias con uno tradicional, dado que la capa intermedia oculta los pormenores de la red de telecomunicaciones. Sin embargo, esta nueva tendencia en los laboratorios remotos abre las puertas a incrementar la potencialidad y versatilidad del sistema, dado que ahora las partes constituyentes del mismo no necesariamente deberán estar emplazadas en un mismo sitio sino que pueden distribuirse en zonas geográficamente alejadas. Además, el usuario puede estar haciendo y rehaciendo el modelo de control desde su propia máquina y no sólo cambiar parámetros de un modelo ya definido en el controlador de la planta.

2.3. La importancia de los laboratorios remotos en la enseñanza de la ingeniería

El Laboratorio Remoto como medio didáctico ofrece una opción diferente del uso de Internet para la educación. Según [10, 11], dentro de las posibilidades que Internet ofrece a la educación se distinguen tres posibles ámbitos de comunicación:

- Foros de discusión: donde gracias a un moderador capacitado puede llevar a cabo debates interesantes en tiempos asincrónicos o diferidos.
- Chats: Conversación entre personas o grupos sin moderador, en tiempo sincrónico (on-line) y generalmente sin agenda previamente establecida.
- Correo electrónico: En modo de tiempo diferido puede intercambiarse texto y otros recursos. Este medio está masificado.

Si bien estos ámbitos están difundidos, actualmente existen otras herramientas de la web 2.0 (entendamos la Internet con características añadidos) que pueden, con planeamiento adecuado servir para la educación (Ver figura 2.3.1). Entre ellas están[12]:

- Las wikis: Son un espacio web corporativo y organizado donde varias personas elaboran contenidos de manera asíncrona. Basta pulsar el botón "editar" para acceder a los contenidos y modificarlos. Suelen mantener un archivo histórico de las versiones anteriores y facilitan la realización de copias de seguridad. Existen diversos servidores de wikis gratuitos.
- Los blogs: Un blog es un espacio web personal en el que su autor/es puede/n escribir cronológicamente artículos, noticias y demás contenidos y recibir realimentación (opiniones, propuestas y críticas) de parte de sus lectores.
- Redes sociales: Son sitios web donde cada usuario tiene una página donde publica contenidos y se comunica con otros usuarios.
- Entornos para compartir recursos en la nube: Constituyen lugares que nos permiten almacenar recursos o contenidos en Internet, compartirlos y visualizarlos cuando nos convenga. Constituyen una inmensa fuente de recursos y lugares donde publicar materiales para su difusión mundial. Existen servicios variados, desde almacenamiento de datos hasta servicios de encuentas en línea y aulas virtuales.

Figura 2.3.1.: Servicios de la web 2.0 [13]

Todas estas posibilidades pueden ser utilizadas para ampliar la experiencia educativa del alumno de carreras de ingeniería y lograr que perciba los conceptos de manera familiar y acorde con sus costumbres en el uso de la informática e internet. Es entonces, dentro de este mundo de opciones, que ahora se pueden estar implementando actividades relacionadas con la experimentación a distancia (gestionadas vía web) que permitan un proceso educativo distinto, moderno y de interés para la juventud de hoy.

Se han puesto en marcha varios proyectos de laboratorios remotos. Cítese a modo de ejemplo tres. En primer lugar, considérese la propuesta pedagógica de utilización de Laboratorios Remotos en carreras de Ingeniería llevada a cabo por UTN Facultad Regional Rafaela con alumnos de las carreras de Ingeniería Electromecánica e Ingeniería Industrial que cursan Física II, en conjunto con la Facultad de Ingeniería Química de la UNL en la ciudad de Santa Fe [11]. Se propuso una clase teórico-práctica de laboratorio con utilización de laboratorio remoto para el estudio del comportamiento en régimen transitorio de circuitos RC, RL y RLC alimentados con corriente continua. El informe cualitativo que se presenta es que hay buena aceptación del alumnado, no se presentan problemas técnicos y se pueden hacer comparaciones exitosas entre experiencias reales y simuladas. Algo que se destaca claramente en el informe de los autores es que la interfaz web es de fácil acceso de manera que no se presentan excesivos requisitos técnicos a los alumnos para operar con el sitio y así los estudiantes están en perfectas condiciones de manejar la experiencia de forma autónoma.

Un segundo caso es el Laboratorio Remoto de la Universidad de Tecnología de Sydney, el cual permite a los alumnos realizar controlar equipamiento de laboratorio real (no simulaciones) y llevar a cabo experimentos de interés académico en cualquier momento del día y a distancia[1]. Dentro del catálogo de experiencias disponibles se cuentan: tanques acoplados, energía hidroeléctrica, robótica, entre otros. Estas experiencias permiten la teleoperación de instrumentos reales en una experiencia didáctica fluída, dado que se permite una monitorización de los procesos mediante camáras conectadas apropiadamente a Internet.

Un tercer ejemplo es el Internet School Experimental System - iSES[2]. En este laboratorio, el usuario puede realizar la adquisición remota de datos reales en tiempo real, además de hacer procesamiento de datos y control de experimentos y otros procesos. En el sitio se presentan varios escenarios posibles a controlar, cada uno con una introducción, descripción del experimento y propuestas de análisis y trabajos prácticos.

Como puede notarse, la Internet de hoy junto con las tecnologías de la web 2.0 permite conectar conectar usuarios con aplicaciones y computadores entre sí de tal manera de brindar una experiencia educativa interesante y pedagógicamente viable.

Hay que reconocer que a veces es necesario romper con paradigmas de enseñanza fuertemente arraigados. Sin embargo, el tomar conciencia de que las prácticas tradicionales de laboratorio experimental pueden ser potenciadas mediante añadir la operación remota vía internet tiene mucho sentido en la sociedad actual. Los jóvenes están inmersos en una cultura fuertemente informatizada, de hecho gran parte de su vida pasa hoy por la red...

[1] http://www.uts.edu.au/future-students/engineering/about-engineering/uts-engineering-facilities/remote-laboratory

[2] http://www.ises.info/index.php/en

Es así como cada docente está ante un desafío... Como comenta [10, 11] “Los nuevos entornos comunicacionales (como Internet) también entran en el salón de clase y dan cuenta de las orientaciones o las restricciones de uso de cada docente”.

2.4. Conclusiones

Como se ha podido apreciar en el desarrollo de este capítulo, los laboratorios remotos constituyen una solución viable para la enseñanza de la ingeniería ante las problemáticas comunes (costo, distancia y accesibilidad) vinculadas a las prácticas experimentales tradicionales. Siendo así, existen investigaciones y trabajos variados sobre el uso de una nueva generación de laboratorios remotos que pueda ser usado para conectar diversos actuadores y sensores controlados por un lazo través de la red.

Además, en el análisis se mencionó también la importancia de un diseño de interfaz de usuario apropiada que resulte no sólo atractiva, sino fácil de manipular y pedagógicamente correcta.

Sin lugar a dudas la correcta implementación, puesta a punto y mantenimiento de un Laboratorio Remoto, como aplicación de uso de Internet, requiere de esfuerzo, gran planeamiento y dedicación. Sin embargo, puede notarse que constituye una propuesta novedosa para la enseñanza de la ingeniería.

Dentro del contexto de este trabajo de tesis, se procede a continuación a describir las tecnologías necesarias involucradas en el diseño de laboratorios remotos.

CAPÍTULO III TECNOLOGÍAS PARA EL DISEÑO DE LABORATORIOS REMOTOS

3.1. Introducción

En varias industrias (entre ellas la robótica) los desarrollos conllevan la necesidad de integración de varias plataformas tecnológicas, junto con la utilización de múltiples lenguajes de programación y una exigencia de alta confiabilidad en el desempeño [14][15]. En virtud a esto, se ve la necesidad de tener una herramienta o arquitectura de software que brinde a los desarrolladores la posibilidad de concentrarse en los objetivos finales de sus proyectos independizándose de los detalles del hardware de bajo nivel. Así, se llega al concepto de Middleware (sistemas de software de capa intermedia).

En este capítulo se comenzará analizando qué son los sistemas de capa intermedia (middleware), su taxonomía (clasificación) y algunas características generales. Posteriormente se muestra qué tipo de arquitectura resulta apropiada para utilizarse en laboratorios remotos. Finalmente se establece cuál de todas las tecnologías de capa intermedia se implementará en el caso de estudio planteado en este trabajo de tesis.

3.2. Middleware: Taxonomía y Definiciones

Una definición general para Middleware es un software que asiste a una aplicación para interactuar o comunicarse con otras aplicaciones, redes, hardware y/o sistemas operativos. Sin embargo, en el contexto de los sistemas distribuidos, la definición se puede enunciar así [16]:

> Middleware: software que provee servicios más allá de los proporcionados por el sistema operativo que permite a los diversos componentes de un sistema distribuido comunicar y gestionar los datos.

Ésta plataforma simplifica el trabajo de los programadores en la compleja tarea de generar las conexiones que son necesarias en los sistemas distribuidos [16]. Los frameworks de capa intermedia están diseñados para enmascarar algunos tipos de heterogeneidad con los que tienen que lidiar los programadores de sistemas distribuidos, entre los cuales se

cuentan: detalles sobre redes, sistemas operativos, hardware, e incluso lenguajes de programación [17]. En la figura 3.2.1 se muestra un gráfico muy claro de la utilización del middleware en el contexto de los sistemas distribuidos.

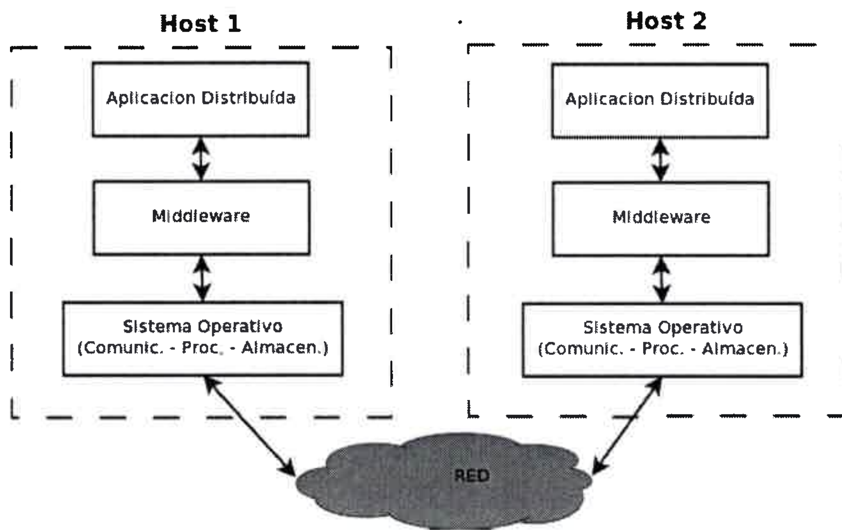
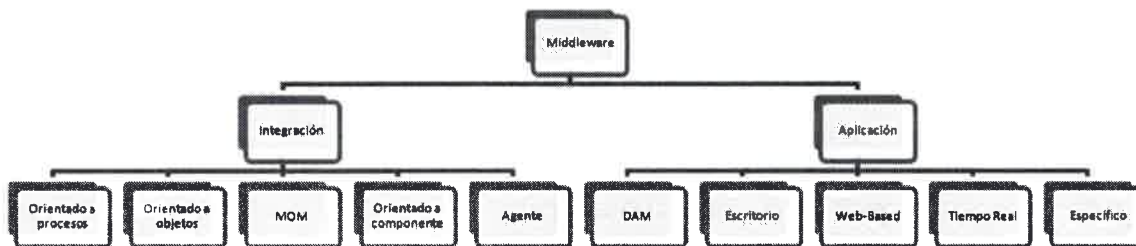


Figura 3.2.1.: Middleware y sistemas distribuidos ([18])

Los software de capa intermedia pueden clasificarse en dos grandes categorías: integración y aplicación. El cuadro 3.2.1 (de [16]) muestra la clasificación completa.



Cuadro 3.2.1.: Taxonomía de Middlewares [16]

Los Middleware orientados a integración incluyen la *capacidad de unirse* con sistemas heterogéneos. Además, generalmente poseen diferentes protocolos de comunicación. Por otro lado, los orientados a aplicaciones son aquellos *ajustados específicamente* para alguna aplicación.

En el primer grupo pueden encontrarse:

- Orientados a procesos: sincrónico, estructura cliente servidor, permite comunicación entre procesos. Ventaja: pueden retornar respuesta aún ante problemas de red y pueden manejar muchos tipos de formatos para datos; Desventajas: no poseen escalabilidad; procesos rígidos.

UNIVERSIDAD NACIONAL DE RIO CUARTO
BIBLIOTECA

- Orientados a objetos: Soportan pedidos de objetos distribuidos. La comunicación entre los objetos puede ser sincronizada o no. Soportan múltiples pedidos similares realizados por múltiples clientes en una transacción. Existen 3 partes que comunican al servidor y el cliente entre si: el proxy, el agente y el proxy remoto. El primero ordena (Marshalling[1]) la información y la transmite a través del agente. Éste es el punto medio de la comunicación y el responsable de contactar con las diversas fuentes de la información y manejar las identidades. El proxy remoto recibe la información (unmarshalling) y la da al servidor. Para enviar la respuesta, el camino es obviamente el inverso. Ventajas: escalables; trabajan con múltiples tipos de información; soporta procesos múltiples. Desventajas: eventualmente necesidad de código extra (wrapper); existencia de vínculos antes de la ejecución.

- Middleware Orientados a Mensajes (Message Oriented Middleware, MOM): Asincrónicos. Mediante este tipo de middleware las transacciones o notificaciones de eventos se llevan a cabo entre los subsistemas o componentes distribuidos mediante mensajes (con cierto formato preestablecido). Este tipo de plataforma permite la creación de sistemas con cohesividad flexible. Desventaja: sobrecarga de red y procesado más lento del lado del servidor [19]. Más adelante se adentra en detalles (sección 3.3).

- Orientados a componentes: El middleware en este caso es una configuración de componentes (programas específicos). Los puntos fuertes de este middleware es que es configurable y reconfigurable.

- Agente: Multicomponente (Entidad - Medio de comunicación - Leyes de interacción). Ventaja: Capaces de manejar múltiples tareas; desventaja: Complejidad alta.

En el segundo grupo:

- DAM (Data Access Middleware): su característica principal es que pueden interactuar con diversas fuentes de datos. En este tipo de middleware se encuentran los que procesan transacciones, gateways de bases de datos y sistemas distribuidos de transacción/procedimiento.

- Escritorio: Los middleware de escritorio pueden hacer variaciones en la presentación de la información pedida por el usuario y otras operaciones de fondo, entre ellas las relacionadas con servicios de transporte y protección de datos, manejo de gráficas e incluso servicios de cifrado y control de accesos.

- Basado en web: Este tipo de middleware asiste al usuario con la navegación web, por ejemplo permitiendo encontrar páginas de su interés y detectar cambios de interés en el usuario basado en su historial de búsquedas. Un ejemplo de middleware

[1]Referido a informática, "marshalling" es el proceso de transformar la representación en memoria de un objeto a un formato de dato suceptible a ser almacenado o transmitido [21].

muy relacionados con las redes son los servidores de aplicaciones. También están los sistemas relacionados con el comercio electrónico y los relacionados con los sistemas móviles y wireless. Estos últimos, aunque no implican conexión cableada, proveen al usuario acceso seguro a servicios web como e-mail, calendarios, etc..

- Tiempo real: Los middleware en tiempo real soportan las peticiones sensibles al tiempo y políticas de planificación. Esto se realiza con servicios que mejoran la eficiencia de las aplicaciones de usuario. En estos middleware se tienen en cuenta tiempos límites de operación. Un ejemplo de estos sistemas son los sistemas multimedia, en especial los servicios de video bajo demanda, los cuales requieren manejar conexiones, asegurar el pago y (muy importante) asegurar la calidad de servicio (QoS) del flujo de datos brindado al cliente.
- Específicos: Hay casos en que los middleware realizan una tarea muy específica que no se puede ajustar a las categorías anteriores. Un ejemplo serían middlewares médicos o educativos.

Dada la sucinta descripción anterior es menester contextualizar el presente análisis. En virtud a que nuestro escenario de análisis son los laboratorios remotos con una gran variedad de elementos a interconectar, algunos distribuidos y multiplataforma, se ve la necesidad de una capa intermedia (middleware) de integración. Dentro de éste ámbito se hará incapié en los MOMs (Message Oriented Middleware), resultando particularmente interesante su característica fuertemente *asíncrona*.

Ahora bien, tras todo lo mencionado surgen algunas preguntas de interés: ¿Porqué están tan difundidos los Middleware orientados a mensajes? ¿Qué significa que sean asíncronos? De la amplia variedad de MOMs disponibles, ¿cuál se escogerá para implementar? Estas preguntas se contestan en la siguiente sección.

3.3. Capa Intermedia orientada a Mensajes (MOM)

Procurando satisfacer las demandas de conectividad heterogénea que presentan varias industrias (entre ellas las relacionadas con robótica), a surgido un mecanismo llamado Capa Intermedia Orientada a Mensajes (MOM, por sus siglas en inglés), el cual provee un método de comunicación transparente entre entidades de software distribuidas.

> MOM puede definirse como cualquier infraestrutura de capa intermedia que provee capacidades de mensajería [14]

Un cliente de un sistema de MOM puede enviar y recibir mensajes hacia y desde otros clientes del sistema de mensajería, considerándose a todos como iguales (comunicación entre pares). Cada cliente se conecta a uno o más servidores que actúan como intermediario en la transacción de mensajes. Este tipo de plataforma permite la creación de sistemas

con cohesividad flexible, esto significa que los cambios en una parte de la estructura se pueden llevar a cabo sin la necesidad de modificar otras partes de la misma.

Ahora bien, ¿sería posible utilizar este tipo de infraestructura para implementar un laboratorio remoto (LR)? Tras investigar se llegó a la conclusión que esto era posible y de hecho ya se estaba utilizando de maneras similares y con buenos resultados [20, 21, 22].

3.3.1. ¿Qué implica que un sistema sea asincrónico?

En los entornos de computación distribuída se habla principalmente de dos modelos de interacción entre las partes de un sistema: comunicación sincrónica y comunicación asincrónica.

Se habla de interacción síncrona cuando el llamado de un procedimiento (o función, o método) implica el bloqueo de la secuencia de código que efectúa la llamada hasta que la ejecución se completa. Este tipo de sistema no posee independencia en el control de procesos, más bien se basa en el hecho de esperar el retorno del control desde el proceso que fue llamado.

Por otra parte, se entiende por modelo de interacción asincrónico cuando la secuencia de código que llama a procesos mantiene el control de los mismos, de manera que no existe un bloqueo de ejecución. Este modelo de interacción requiere un intermediario para manejar el intercambio de solicitudes; normalmente el agente intermediario es una cola de mensajes. En la figura 3.3.1 se visualizan esquemas explicativos de ambos modelos.

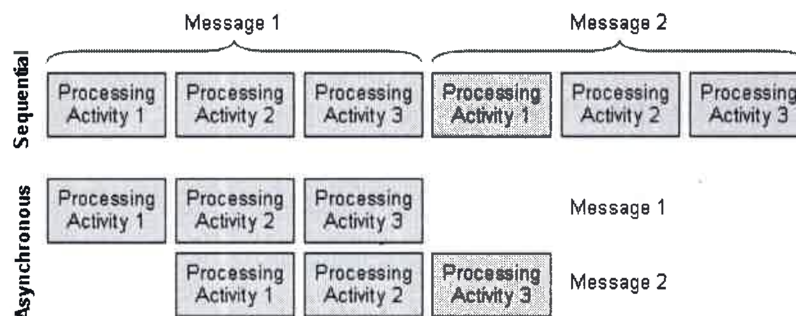


Figura 3.3.1.: Diferencia entre interacción sincrónica y asincrónica [23]

Si bien el último esquema analizado conlleva más complejidad, ésta es justificada por la independencia entre procesos que permite. De esta manera, las partes del sistema pueden seguir trabajando (sin bloqueos o esperas) independientemente del estado de los demás participantes.

A continuación se resumen algunas ventajas de los MOMs.

3.3.2. Ventajas de Middlewares orientados a mensajes

Las principales ventajas de las capas intermedias son[24]:

- Modularidad de software
- Abstracción de arquitectura de hardware: de esta manera se oculta las características específicas del hardware de bajo nivel, dando así a los desarrolladores APIs (del inglés: Application Programming Interface) más convenientes y estandarizadas.
- Independencia de plataforma
- Portabilidad: debe poderse correr en cualquier plataforma con sólo cambiar la configuración del sistema. Por lo tanto, no debe depender de ningún dispositivo o algoritmo de software específico.
- Escalable y actualiable.
- Además, debe ser robusto, confiable, fácil de usar, flexible y soportar programción en paralelo y sistemas distribuídos. También, debe permitir cambiar configuraciones en tiempo real a la vez que brinda algunos aspectos de seguridad, como autentificación, autorizaciones y conexiones seguras.

Tras el análisis previo en este capítulo pueden surgir algunos interrogantes. Por ejemplo: ¿serán los middleware los únicos elementos de integración para sistemas distribuidos y multi-arquitectura? ¿Como son las comunicaciones en los sistemas orientados a mensajes? ¿Cuán "buenos" son estos sistemas? La respuesta a estos interrogantes se encontrará a continuación.

3.4. Comparación entre MOMs y RPC

Se ha mencionado que la integración es una necesidad no solo de la industria sino de las arquitectura de laboratoios remotos. ¿Cómo lograrla? Una manera es la utilización de Middleware orientado a mensajes. En el cuadro 3.2 se hace un resumen de conceptos que resultan importantes de tener en mente [15].

Ahora bien, los middleware no son la única manera de lograr la comunicación intercomponente. Otras opciones de integración son [25]:

- Transferencia de archivos: un componente produce archivos de datos compartidos que otros componentes utilizan
- Base de datos compartida: cada componente lee y graba datos en una base de datos común

Concepto	Explicación
Middleware de mensajeo	Aplicación de terceros que provee la infraestructura y capacidad de mensajeo. Ejemplo: MSMQ (Message Queuing), MS-CCR (Concurrency and Coordination Runtime), ROS (Robot Operating System)
Componente	Aplicación o porción de código independiente que se comunica utilizando el middleware
Systema basado en mensajes	El sistema visto como un todo, incluyendo todos los componentes integrados y el middleware de mensajeo

Cuadro 3.4.1.: Resumen de algunos conceptos

- Invocación de procesos remotos: cada componente declara los procedimientos específicos que deben invocarse remotamente para exponer el comportamiento y el intercambio de datos.

A continuación se introducen algunas características del Procedimiento de Llamada Remota y también se analiza con mayor profundidad los sistemas MOM. Finalmente se hace una breve comparación entre ambos.

3.4.1. Introducción al Procedimiento de Llamada Remota tradicional (RPC)

El objetivo principal de RPC es permitir que dos procesos interactúen. Algunos middleware que lo utilizan son: CORBA, Java RMI, Microsoft DCOM y XML-RPC (usado por el Robot Operation System)[14].

Su funcionamiento permite que los procesos que interactúan entre sí lo hagan como si fueran uno solo. Así, basándose en un modelo síncrono de interacción, RPC es similar a un procedimiento de llamada local donde el control se pasa al proceso invocado de una manera síncrona secuencial mientras el procedimiento que realiza la invocación bloquea su ejecución hasta que recibe repuesta a su llamada. En la figura 3.4.1 (de [14]) se muestra un esquema que ejemplifica el funcionamiento del RPC.

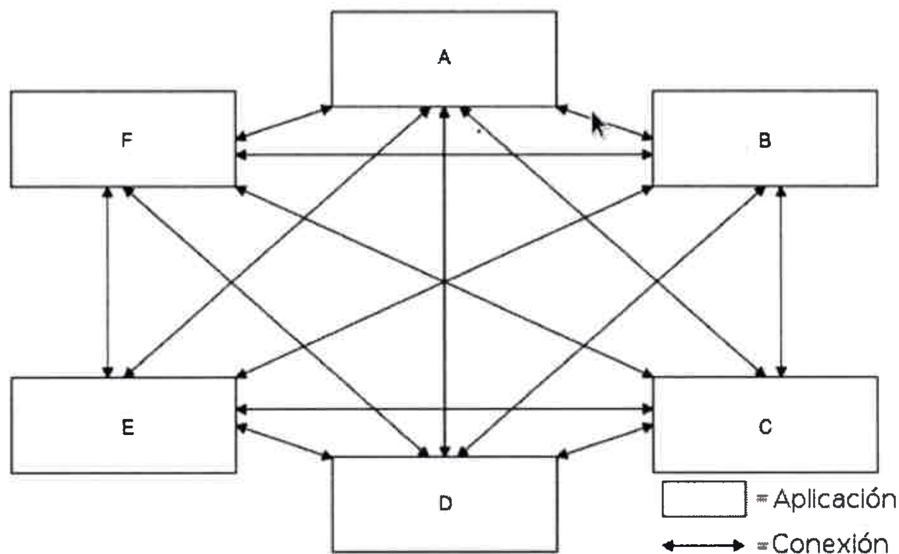


Figura 3.4.1.: Esquema de funcionamiento de RPC [14]

3.4.1.1. Algunos detalles sobre RPC

RPC está diseñado para trabajar con un modelo de sistema fuertemente acoplado, de tal manera que cualquier cambio en las interfaces debe ser propagado inmediatamente a las partes. Esta característica hace de RPC un sistema muy invasivo en lo que se refiere al mecanismo de distribución de información. Así, mientras más interfaces haya o más cambios se realicen, el costo en ancho de banda se incrementa, dado que todas las partes se deben enterar de los mismos.

La confiabilidad es otra de las características en un sistema a la que se da mucha atención; se requiere que el transporte de datos entre las partes del sistema se pueda lograr con "garantías" o fiabilidad. A este respecto, puede decirse que RPC, en la mayoría de sus implementaciones, no provee garantía de confiabilidad en la capacidad de comunicación.

Por otro lado, en virtud a que en el modelo RPC los subsistemas son interdependientes, se requiere que todos estén con disponibilidad de funcionamiento para que el sistema trabaje correctamente. También una falla en un componente puede generar la falla general del sistema.

3.4.2. Introducción a MOM

Los sistemas MOM proporcionan comunicación distribuida sobre la base del modelo de interacción asíncrona, resolviendo así muchas de las limitaciones que se encuentran en RPC. Los participantes en un sistema basado en MOM no están obligados a bloquear y esperar a enviar un mensaje, se les permite continuar con el proceso una vez que un

UNIVERSIDAD NACIONAL DE RIO CUARTO
BIBLIOTECA

mensaje ha sido enviado. Esto permite la entrega de mensajes cuando el remitente o el receptor no está activo o disponible para responder en el momento de la ejecución.

En los sistemas distribuidos basados en MOM se ofrece una comunicación entre procesos orientada a la prestación de servicios. La figura 3.4.2 (extraída de [14])muestra un esquema de comunicación basada en capa intermedia orientada a mensajes .

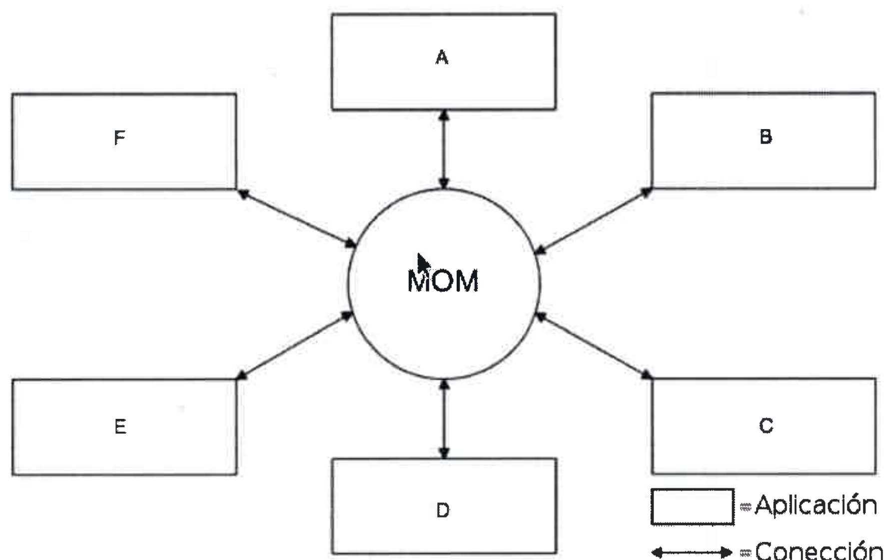


Figura 3.4.2.: Esquema de sistema MOM [14]

3.4.2.1. Algunos detalles sobre MOM

Dado que MOM inserta una capa (que sirve de intermediario) entre las partes que intercambian mensajes, se pierde necesidad de acoplamiento, lo que resulta en un sistema más cohesivo y donde no es necesario hacer tantas adaptaciones entre fuentes y destinos de mensajes. Esto es una ventaja muy importante.

Además, MOM tiene una confiabilidad alta y, generalmente configurable, en cuanto a la persistencia de datos, dado que utiliza un mecanismo de almacenamiento y reenvío. Este sistema resulta útil cuando ocurren pérdida de mensajes en la red de comunicaciones o cuando alguna parte del sistema está ocupada o no disponible. También, los sistemas MOM son capaces de garantizar que los mensajes se entregarán y lo harán sólo una vez (sin repeticiones o "ecos") al destinatario.

En cuanto al crecimiento del sistema, MOM permite que sus partes sean actualizadas o mejoradas independientemente. También, brinda al sistema la capacidad de hacer frente a picos de actividad en un subsistema sin afectar al resto. Dicho equilibrio en la carga se mantiene dado que MOM permite a un subsistema recibir un mensaje cuando esté listo para ello, en vez de forzarlo.

También cabe aclarar que la característica de acoplamiento débil que tiene MOM permite que la disponibilidad del sistema no dependa de la disponibilidad de todas las partes, de tal manera que el conjunto se hace menos sensible a los fallos. Además, ya que el acoplamiento exige menos compatibilización entre las fuentes y los destinos de mensajes (y obviamente menos sobrecarga a la red de comunicación) se mejora el tiempo de respuesta del sistema.

3.4.3. Cuándo usar MOM y cuando RPC

Tanto MOM como RPC tienen sus ventajas y desventajas. Se encuentra una buena comparación entre ellos en [14].RPC provee un método directo de mensajeo utilizando el modelo de interacción sincrónica. También debe destacarse la garantía de procesamiento secuencial de los mensajes; RPC es consistente y lleva a cabo su trabajo en el orden correcto. Si la integridad temporal de los datos es más importante que el desempeño global, RPC es el modelo apropiado a considerar.

Sin embargo, RPC tiene problemas asociados con el ancho de banda, la inflexibilidad y el acoplamiento fuerte entre las partes. Además, ante fallos o indisponibilidad de algún subsistema todo se vuelve disfuncional; incluso la sobrecarga de alguna sección afecta el rendimiento global del sistema.

Entonces, puede decirse que el RPC tradicional es un modelo simple y muy apropiado para el paradigma de un cliente comunicándose con un servidor; el RPC tradicional no tiene soporte nativo para comunicaciones uno a varios.

Por otro lado, MOM simplifica el proceso de construir sistemas empresariales altamente flexibles y distribuidos; incluso el ancho de banda para comunicaciones requerido es menor que en una implementación similar basada en RPC. También, los Middleware orientado a mensajes permiten a los sistemas evolucionar sin grandes modificaciones en las aplicaciones, proveyendo así una infraestructura que acomoda los cambios funcionales en el tiempo sin compromenter o afectar el desempeño del sistema y su escalabilidad. Así, en este tipo de sistema se soporta un gran número de generadores y consumidores de mensajes.

Sin embargo, los sistemas MOM, en virtud a su modelo de interacción asincrónico, no pueden garantizar la exactitud temporal de los datos. No hay control sobre el orden en que se procesarán los datos.

Entonces, puede decirse que si lo requerido es un sistema geográficamente disperso una red de comunicaciones de poca capacidad y se pretende confiabilidad, flexibilidad y calidad, entonces los Middleware basados en Mensajes son la solucion apropiada.

Finalmente, dado que un laboratorio remoto cumple con las premisas expuestas en el párrafo anterior, se ve apropiada la utilización de MOM como modelo del sistema.

3.5. Modelo de intercambio de mensajes en los MOM

Los sistemas MOM tienen por lo general dos tipos bien diferenciado de intercambio de mensajes[2]:

- Punto a Punto: Provee un intercambio asincrónico directo de mensajes entre entidades de software vía una cola (generalmente tipo FIFO[3]). Cada mensaje es brindado una sola vez y a un solo cliente y, si bien multiples consumidores pueden leer de la misma cola, cada mensaje podrá ser leído por uno de ellos. Esta técnica permite el balance de cargas en el sistema. En este modelo de mensaje, los mismos son siempre transmitidos a destino y cada uno será almacenados en la cola hasta que el consumidor esté listo para recibirlo.
- Publicador / Suscriptor: Este es un mecanismo muy poderoso para hacer difusión de mensajes entre consumidores y productores anónimos. De hecho, bajo este modelo está permitida la distribución de mensajes uno a muchos u muchos a uno. Es interesante destacar que niguna de las partes necesitan tener conocimiento holístico del sistema. Así, cuando desea transmitir, el publicador produce un mensaje dentro de un canal (llamado "tópico") al cual se suscribirá todo aquel que esté interesado en recibir los mensajes. Este proceso se da dentro de un motor o núcleo de publicaciones o subscripciones.

En la Figura 3.5.1 se muestran los dos modelos de mensajes analizados.

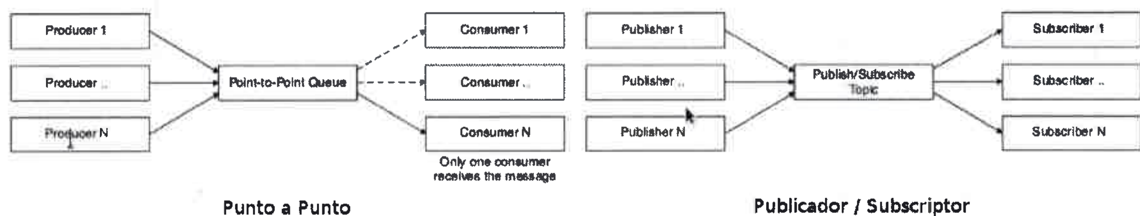


Figura 3.5.1.: Modelos de intercambio de mensajes [14]

3.5.1. Nombres jerárquicos de tópicos

Los canales jerárquicos (o "tópicos") son un mecanismo de agrupación de destino en el modelo de mensajería publicador/suscriptor [14]. Este tipo de estructura permite que los canales se aniden bajo otros canales según detalle. Cada subcanal ofrece una selección más granular de los mensajes contenidos en su matriz. Los clientes de los canales jerárquicos pueden suscribirse al nivel más apropiado de canal con el fin de recibir los mensajes

[2]Entiéndase por mensaje la información (con cierta estructura) que una de las partes del sistema envía a otra. En la sección 3.6 se analiza con más profundidad este tema.

[3]FIFO=First Input First Output. Esto implica que los mensajes se almacenan y luego se extraen ordenadamente (el primero en entrar a la espera será el primero en despacharse).

más relevantes para ellos. En los sistemas de gran escala, la agrupación de mensajes en tipos relacionados (por ejemplo, en los canales) ayuda a gestionar grandes volúmenes de diferentes mensajes.

Cabe aclarar que es un requisito indispensable que el esquema de espacio de nombre de los canales sea bien definido y conocido por las partes intervinientes del sistema. Asimismo, es responsabilidad del publicador escoger el canal en el que publicará; el receptor debe ser capaz de buscar dentro de la estructura jerárquica de tópicos y escoger a cuál se suscribirá.

3.5.2. Comparación entre ambos modelos de mensajeo

Los dos modelos tienen capacidades muy similares y puede utilizarse una combinación de ambos [14]. La diferencia fundamental entre los modelos se reduce al hecho de que dentro del esquema de publicación/suscripción todos los consumidores a un tópico (o canal) recibirá un mensaje publicado en el mismo, mientras que en el modelo de punto-a-punto un solo consumidor recibirá el mensaje. El modelo publicación/suscripción se utiliza normalmente en un escenario de emisión cuando un emisor desea enviar un mensaje a los clientes sin tener un control real sobre el número de clientes que reciben el mensaje y sin garantía de recepción exitosa. Este modelo de mensajería tiene como ventaja la flexibilidad; su principal desventaja es su complejidad.

Por el contrario, en el modelo punto-a-punto, aunque muchos consumidores pueden estar escuchando en una cola, sólo uno recibe cada mensaje. En este modelo hay garantía de que un consumidor recibirá el mensaje cuando esté listo y lo recibirá una sola vez. Si bien es cierto que este modelo es menos flexible, su ventaja es la simplicidad.

3.6. Estructura de Mensajes

Cuando se pasan datos entre dos procesos con espacios de memoria separados, los datos deben ser empaquetados en un "mensaje" con un formato que el receptor será capaz de dessempaquetar y entender [15]. El remitente del mensaje pasa el mensaje a través de un canal de mensajes. El receptor recupera el mensaje del canal de mensajes y transforma el mensaje en las estructuras de datos internas apropiadas para la tarea en cuestión.

Un mensaje se compone de dos partes fundamentales:

- Cabecera: Describe detalles del mensaje (Ej.: estampa de tiempo, tiempo de expiración, origen, etc.).
- Cuerpo: El contenido o datos “útiles” para el receptor.

También, a la hora de mandar un mensaje, el que lo envía puede tener una determinada “intención” o propósito:

- Mensaje de comando: invoca un procedimiento en el receptor.
- Mensaje de documentación: pasa datos al receptor.
- Mensaje de evento: notifica un cambio de estado al receptor.
- Mensaje de respuesta: envía una respuesta a la aplicación receptora.

Obviamente es muy importante que todos los que intervienen en la comunicación acuerden un formato de mensaje de manera que haya interoperabilidad entre las partes, mejor reuso de componentes y facilidad en la extensión y escalado del sistema.

3.7. Servicios comunes de los MOMs

Las plataformas orientadas a mensajes tienen algunos servicios que resultan importantes para su valoración y utilización. A continuación se analizan algunos de estos [14].

3.7.1. Filtrado de mensajes

El filtrado de mensajes permite a un receptor ser selectivo sobre los mensajes que tomará de un determinado canal (tópico). El filtrado suele operar tanto en las propiedades del mensaje como sobre su contenido.

3.7.2. Transacciones

Las transacciones (conjunto de órdenes) posibilitan la agrupación de las tareas en una unidad de trabajo. Las características de una transacción se resumen en el cuadro 3.3.

Características	Descripción
Atómica	Todas las tareas se deben completar o ninguna se debe completar.
Consistente	Si el estado inicial es consistente, el final también lo será independientemente del resultado de la transacción (éxito o falla)
Aislada	Cada transacción se debe ejecutar de forma independiente sin afectar a otras.
Durable	El efecto de una transacción confirmada no se debe perder por una falla subsecuente de un proveedor u otro agente.

Cuadro 3.7.1.: Características de una transacción

Un MOM tiene la capacidad de incluir un mensaje que está siendo enviado o recibido dentro de una transacción. Es importante recordar que los mensajes son entidades autó-

nomas autocontenidas. A continuación se analizan las características transaccionales del paso de mensajes.

El paso de mensaje como parte de una transacción se utiliza cuando se desea llevar a cabo varias tareas de mensajería de manera que todas tendrán éxito o fracasarán. Cuando se utiliza la mensajería transaccional, la aplicación de envío o de recepción tiene la oportunidad de confirmar la transacción (todas las operaciones han tenido éxito), o cancelar la operación (una de las operaciones fallado). Si se anula una transacción, todas las operaciones se revierten al estado inicial. Los mensajes entregados al servidor en una transacción no se reenvían al cliente receptor hasta que el cliente que envía confirma la transacción. Las transacciones pueden contener varios mensajes y es digno de mención que también pueden utilizarse colas para realizarlas.

En un paso de mensaje dentro de una transacción se pueden distinguir tres roles bien diferenciados: El productor, el agente servidor de mensajes (broker), receptor. En la figura 3.7.1 se esquematizan el papel de cada uno en una operación típica.

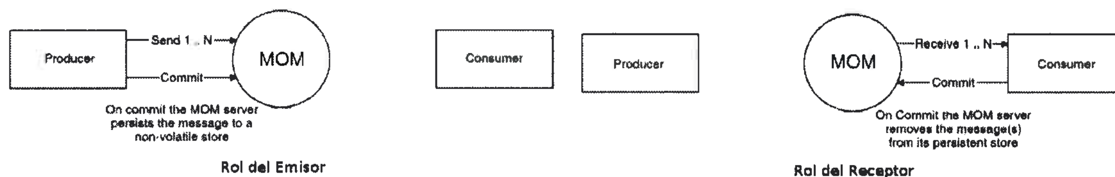


Figura 3.7.1.: Roles en una transacción [14]

El productor es el encargado de enviar el/los mensaje/s al broker. Éste último almacena la información y envía los mensajes al receptor tras la confirmación del emisor.

Por otro lado, el receptor acepta los mensajes del broker. Si hay necesidad de reenvío el broker así lo hace; cuando la operación es aceptada y terminada el broker puede disponer de los mensajes almacenados (por ejemplo borrarlos).

Cabe mencionar que la entrega de mensajes suele tener varias características de QoS (calidad de servicio) configurables, entre ellas el número de intentos ante falla y el tiempo de vida de los paquetes.

3.7.3. Garantías de entrega de mensajes

Para garantizar la entrega de los mensajes en las plataformas MOM, estos se almacenan en una memoria no volátil (ej.: disco duro) hasta que el consumidor confirma la recepción de los mismos. Si la confirmación se demora más de un tiempo razonable, el servidor reenvía la información.

3.7.4. Formatos de mensajes

Dependiendo de la implementación de MOM que se considere, se puede hallar disponibles distintos tipo de formato de mensajes. Algunos tipos comunes son: Texto (incluyendo XML), Objeto, Flujo de Datos, Mapas Hash, Flujo Multimedia, etc. Algunos MOM permiten la transformación de un mensaje en cierto formato a otro.

3.7.5. Balance de carga

El balance de carga es el proceso de repartir la carga del sistema sobre un número dado de servidores (máquinas). Si el sistema funciona correctamente debe distribuir el trabajo, asignando dinámicamente el mismo al dispositivo con la carga más ligera. Existen dos modelos de balance de carga. El primero (modelo "push"), usa un algoritmo para balancear la carga sobre múltiples servidores. El segundo modelo (llamado "pull") consiste en colocar los mensajes entrantes en una cola punto a punto y los terminales que consumen entonces pueden sacar mensajes de esta cola a su propio ritmo. Esto permite un balance de carga real, dado que cada servidor sólo sacará un mensaje desde la cola una vez que es capaz de procesarlo.

3.7.6. Agrupamiento (Clustering)

Clustering es la distribución de una aplicación a través de múltiples servidores para escalar más allá de los límites, tanto de rendimiento y fiabilidad, de un solo servidor. Cuando se han alcanzado los límites del software de servidor o de los límites físicos del hardware, la carga debe repartirse entre varios servidores o máquinas para escalar aún más el sistema. El agrupamiento permite distribuir sin problemas a través de múltiples servidores/máquinas y todavía mantener una única entidad lógica y una interfaz virtual para responder a las peticiones de los clientes. La agrupación de clústeres crea implementaciones altamente escalables y confiables y reduce al mínimo el número de servidores necesarios para hacer frente a grandes cargas de trabajo.

3.8. Desempeño de los sistemas basados en mensajes

Probablemente sea una suposición común pensar que el desarrollo de un sistema basado en mensajes tiene un enorme costo para el rendimiento [15]. Objetos de dominio se convierten en mensajes, los mensajes se pasan a través de canales, routers y filtros para interceptar y reenviar mensajes y los mensajes se convierten en objetos de dominio... esto suena como un cantidad enorme de cosas que hacer. Pero debido a que cada componente

se ejecuta en su propio proceso o subproceso, no necesita esperar a que otros componentes terminen para completar su trabajo antes de poder pasar a manejar otro mensaje o tarea. Ésa es una diferencia fundamental entre el procesamiento síncrono y el asíncrono. La figura 3.3.1 (ya mostrada) presenta esta comparación [25][23].

Puede notarse en la antedicha figura que el proceso secuencial del procesamiento sincrónico conlleva la demora de que antes de pasar a procesar un mensaje es requisito que se haya terminado de trabajar con el anterior. En cambio, en el modelo asincrónico se puede empezar a procesar el siguiente mensaje cuando el primero haya sido procesado en parte. Esto compensa la sobrecarga adicional impartida por esta estructura de mensajería.

Por otro lado, se puede decir que dentro de una fortaleza de los Middleware orientados a mensajes radica en su capacidad para almacenar, enrutar y transformar los mensajes mientras son transportados desde los emisores hasta los receptores. A su vez, su particular deventaja radica en la necesidad de un agente para transferencia de mensajes (broker) el cual significa un componente más en el sistema y, consecuentemente, más complejidad y gasto de mantenimiento. Además, cabe aclarar que dado el funcionamiento intrínsecamente asincrónico de los MOMs estos pueden no ajustarse bien en algunas situaciones donde se exija respuestas en tiempo real.

Para la aplicación en el contexto de los laboratorios remotos es posible utilizar MOMs. Algunos trabajos que establecen este hecho y muestran resultados de aplicación son [26, 27, 28, 29, 30] Puede apreciarse en ellos que la implementación implica retardos tolerables menores a 25[ms], lo que según [3] corresponde a una cota para laboratorios remotos.

Además, el carácter asincrónico del funcionamiento de los MOMs los convierten en una opción apropiada para permitir que cada parte (sensor, controlador, etc.) del laboratorio remoto pueda responder a las solicitudes a su ritmo sin paralizar necesariamente el funcionamiento de todo el sistema.

3.9. Conclusiones

En el desarrollo de este capítulo, se ha presentado las tecnologías implicadas en el diseño de los laboratorios remotos, centrando la atención en los Middleware Orientados a Mensajes (MOM, por sus siglas en inglés).

Además, el análisis incluyó una descripción general de las características de los MOM y su funcionamiento genérico. También se presentó una sucinta comparativa con otra tecnología utilizada en los sistemas distribuídos, a saber RPC. De dicha comparación se destaca que el carácter asincrónico de los MOMs permite aplicaciones donde cada unidad puede operar y responder a peticiones a su ritmo sin necesidad de paralizar el sistema

entero. Dicha propiedad reviste especial interés en el desarrollo de la presente temática de tesis.

Finalmente se comenta que para la realización de un caso de estudio de laboratorio remoto se escogerá un MOM en particular: el ROS (Robot Operating System). Este es un desarrollo del Laboratorio de Integligencia Artificial de Stanford que ahora es continuado por el instituto Willow Garage. La elección de esta plataforma de software se vio motivada por sus características de rendimiento y por la capacidad de trabajar con lenguaje python. Además, la documentación disponible es muy buena y la cantidad de módulos, ejemplos didácticos y paquetes disponibles para trabajar es abundante. A continuación, dentro del contexto de este trabajo de tesis, se procede a describir las características de dicho middleware y su funcionamiento.

UNIVERSIDAD NACIONAL DE RIO CUARTO
BIBLIOTECA

CAPÍTULO IV LA CAPA INTERMEDIA: ROBOTIC OPERATION SYSTEM (ROS)

Hoy en día los laboratorios remotos se han vuelto una herramienta útil en la enseñanza de las ciencias y la ingeniería. Diversos factores (económicos, edilicios, geográficos, entre otros) junto a la disponibilidad de conexiones a Internet de velocidad aceptable han contribuído a pensar en la posibilidad de que el alumnado pueda compartir experiencias de laboratorio a distancia. Es en este marco donde se encuentra la necesidad de contar con un middleware, o capa intermedia, que permita abstraerse de las cuestiones relacionadas con los parámetros y características intrínsecas de la red para centrarse en el proceso a controlar y monitoriear en sí. Por tanto, desde un punto de vista educativo, cabe aclarar que un estudiante que opera un sistema remotamente no debería ver la diferencia con respecto a la operación local del mismo, lo cual sugiere la importancia de escoger bien el middleware a utilizar. En el presente trabajo se utiliza el Robot Operation System (ROS) y se evalúa su desempeño. Esta elección se fundamenta en tres aspectos: las prestaciones de esta arquitectura de software, la calidad de documentación y repositorios, y la buena compatibilidad con el lenguaje de programación Python. A medida que se desarrolle este capítulo cada aspecto resultará ampliado.

A continuación se describen algunas características del sistema ROS y su funcionamiento. Como base de la información se utiliza principalmente la wiki oficial del Robot Operating System: http://wiki.ros.org/. Además, se mencionan ejemplos prácticos y de ejecución de comandos bajo sistema operativo linux (Ubuntu 12.04 con ROS Groovy).

4.1. Aspectos generales

ROS es un meta-sistema operativo de código abierto pensado para robótica que corre sobre plataformas basadas en UNIX. Ofrece los servicios que se esperarían de un sistema operativo, incluida la de abstracción de hardware de bajo nivel de control del dispositivo, la aplicación de la funcionalidad de uso común, de paso de mensajes entre los procesos y de gestión de paquetes. También proporciona herramientas y bibliotecas para la obtención, la construcción, la escritura y la ejecución de código en varios equipos [31]. La wiki oficial presenta mucha información y permite que los usuarios puedan colaborar en la incrementación de la base de conocimientos. La figura 4.1.1 muestra una vista de dicho

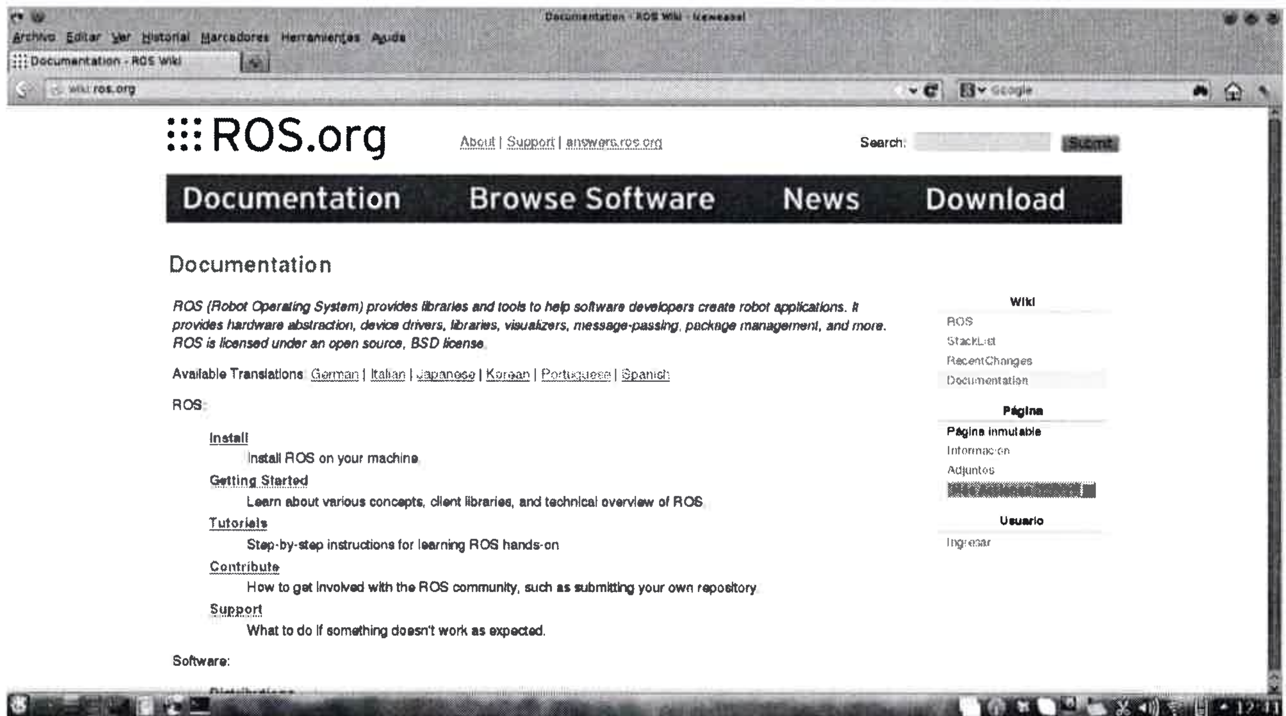


Figura 4.1.1.: Vista del sitio oficial de ROS

sitio de internet. No es un tema menor la disponibilidad de información y el apoyo de la comunidad de usuarios, dado que la calidad de la documentacioń influye directamente en la pendiente de la curva de aprendizaje. La experiencia con la documentación de ROS fue muy satisfactoria.

4.2. Arquitectura del sistema ROS

La filosofía de ROS se puede resumir en pocos items:

- Punto a Punto
- Basado en herramientas
- Multilenguaje
- Modular
- Gratis y de código abierto

Por ejemplo, en los robots de servicio grandes para que ROS fue diseñado, normalmente hay varios ordenadores de a bordo conectados a través de Ethernet (ver figura 4.2.1). Este segmento de la red es puenteado a través de LAN inalámbrica de alta potencia a las máquinas (de escritorio o servidores) que están ejecutando las tareas de computación intensiva (informática, visión, reconocimiento de voz, etc.)

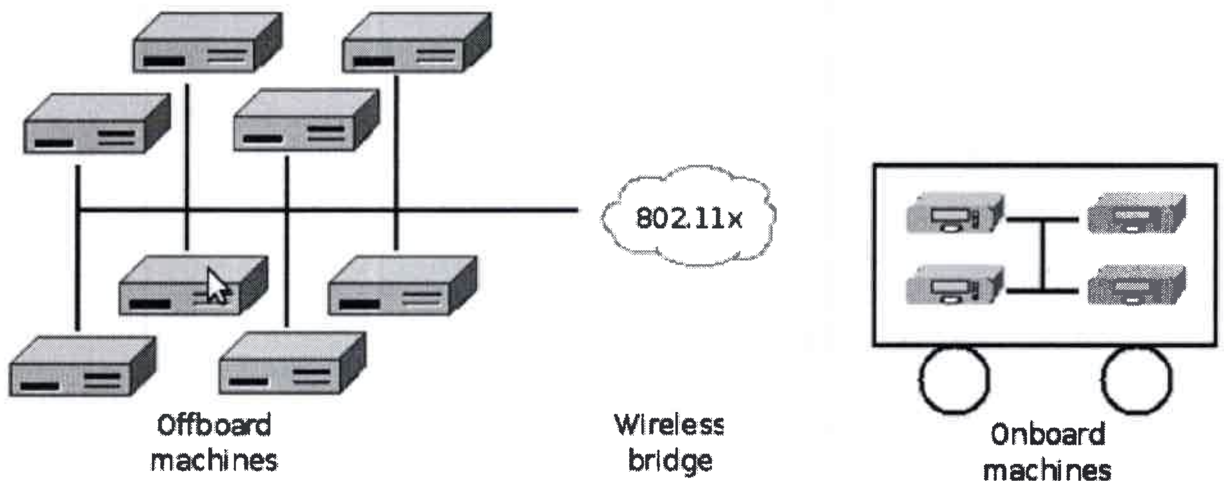


Figura 4.2.1.: Arquitectura general sistema ROS [31]

Por tanto, un sistema construido utilizando ROS consta de un número de procesos, los que podrían estar corriendo en un número de hosts diferentes, conectados en tiempo de ejecución mediante una topología de punto a punto.

Además, el hecho de que este sistema sea neutral en cuanto a los lenguajes permite que los usuarios se sientan más cómodos al programar. Actualmente son soportados cuatro lenguajes: C++, Python, Octave y LISP. Esta flexibilidad en cuanto a lenguaje se basa e el hecho de que ROS trabaja mediante mensajes sobre conexiones punto a punto; esto es mediante un sistema de publicación y registro basado en el protocolo XML-RCP. Para este protocolo hay soporte para la mayoría de lenguajes.

En cuanto a estructura de software, es de interés que sus autores apostaron al diseño de un microkernel con funciones específicas y a la modularización de las demás librerías y drivers. Ellos lo explican diciendo: "Creemos que la pérdida de eficacia es más que compensado por las ganancias en la estabilidad y la complejidad gestión"(original es en inglés).

Finalmente, es interesante destacar que este desarrollo está liberado bajo licencia BSD, la cual permite su uso para desarrollos con fines tanto comerciales como no comerciales. Este hecho permite que haya muchos desarrolladores independientes que puedan trabajar con esta herramienta sin necesidad de grandes inversiones.

4.3. Conceptos de la arquitectura ROS

Algunos términos que identifican las partes y diversos conceptos de la arquitectura ROS se exponen a continuación. Cabe aclarar que para el análisis es posible realizar una división conceptual de tres niveles: nivel de archivos de sistemas, nivel de funcionamiento y nivel

de comunidad[32].

NOTA: La descripción subsiguiente se realiza considerando el sistema de compilación ROSBUILD, el cual fue utilizado para el presente trabajo. Esta aclaración es pertinente pues actualmente se está utilizando un nuevo sistema llamado CATKIN. En las secciones Anexo-A y Anexo-B encontrará más información extraída de la wiki oficial.

4.3.1. Nivel de archivos de sistema

Dentro de este nivel identificamos los siguientes conceptos, los cuales identifican los recursos de ROS que se encuentran en el disco duro:

- Paquetes: Son la unidad principal de organización del software en ROS. Puede contener: procesos ejecutables (nodos), librerías dependientes de ROS, conjunto de datos, archivos de configuración, etc.
- Manifiesto de un paquete: Es un archivo .xml que provee información sobre el paquete, por ejemplo licencia y dependencias de otros paquetes.
- Pila (stack): Las pilas son colecciones de paquetes que proporcionan la funcionalidad agregada. Además son la manera en cómo se libera el software ROS y se han asociado los números de versión.
- Manifiesto de una pila: Es un archivo .xml que provee información sobre la pila, por ejemplo licencia y dependencias de otras pilas.
- Tipo de mensajes: Las descripciones de los mensajes se almacenan en un archivo .msg dentro de una subcarpeta "msg" en el paquete y define la estructura del mensaje a enviar en ROS.
- Tipo de mensajes: Las descripciones de los mensajes se almacenan en un archivo .srv dentro de una subcarpeta "srv" en el paquete y define la estructura del servicio en ROS.

4.3.2. Nivel de funcionamiento

- Nodos: Los nodos son procesos que realizan el cálculo. ROS está diseñado para ser modular en una escala de grano fino; un sistema de control del robot comprende generalmente muchos nodos. Por ejemplo, un nodo controla un telémetro de láser, otro controla los motores de las ruedas, otro lleva a cabo la localización, otro realiza planificación de trayectoria, otro nodo proporciona una vista gráfica del sistema, y así sucesivamente. Un nodo es escrito usando una biblioteca de cliente de ROS, tal como roscpp (para usar lenguaje de programación C++) o rospy (para usar lenguaje

de programación Python, algo que se tuvo muy en cuenta dado la experiencia previa que ya se tenía en el dominio de este lenguaje)

- Mensajes: Los nodos se comunican entre sí mediante el paso de mensajes. Un mensaje es simplemente una estructura de datos que comprende la tipificación de los campos; pueden ser tipos estándares (Punto flotante, entero, cadena, etc) o personalizados.
- Tópicos (Tema): Un nodo envía un mensaje mediante su publicación en un determinado tema . El tema es un nombre que se utiliza para identificar el contenido del mensaje. Un nodo que esté interesado en un determinado tipo de datos suscribirá el tema correspondiente. Puede haber múltiples editores y suscriptores concurrentes para un solo tema, y un único nodo puede publicar y / o suscribirse a múltiples temas. En general, los editores y suscriptores no son conscientes de la existencia de los demás. La idea es disociar la producción de información de su consumo. Lógicamente, se puede pensar en un tema como un "canal" de mensajes con un tipo de dato bien definido. Cada "canal" tiene un nombre, y cualquier persona puede conectarse para enviar o recibir mensajes mientras sean del tipo admitido.
- Servicios: El modelo de Publicación / Subscripción es un paradigma de comunicación muy flexible y no resulta apropiado para las interacciones solicitud-repuesta, las cuales son requeridas generalmente en un sistema distribuido. Así, el paradigma Petición / Respuesta se realiza a través de servicios, que se definen por un par de estructuras de mensajes: uno para la solicitud y uno por la respuesta. Un nodo servidor ofrece un servicio bajo cierto nombre y un cliente utiliza el servicio mediante el envío del mensaje de solicitud y *queda a la espera* de la respuesta (bloqueante). Las bibliotecas de cliente ROS (por ejemplo: roscpp o rospy) generalmente presentan esta interacción para el programador como si fuera una llamada a procedimiento remoto (RPC).
- Bolsas: Las bolsas son un formato para guardar y reproducir datos de mensajes de ROS. Las bolsas son un mecanismo importante para el almacenamiento de datos, como los datos de los sensores, que pueden ser difíciles de recoger, pero es necesario para desarrollar y probar algoritmos.
- Maestro: Servicio de nombres para ROS el cual ayuda a los nodos a contactar con sus pares y obtener información útil.
- rosout: Equivalente para ros de la salida estándar (stdout) y la salida estándar de error (stderr).
- roscore: Master + rosout + servidor de parámetros
- servidor de parámetros: Es una especie de diccionario compartido que sirve a los nodos de manera que puedan grabar y acceder a parámetros en tiempo de ejecución.

Utiliza XML-RPC.

Básicamente, dos modos de funcionamiento pueden utilizarse entre nodos: Publicador-Subscriptor y Servidor-Cliente. La figura 4.3.1 muestra un esquema didáctico.

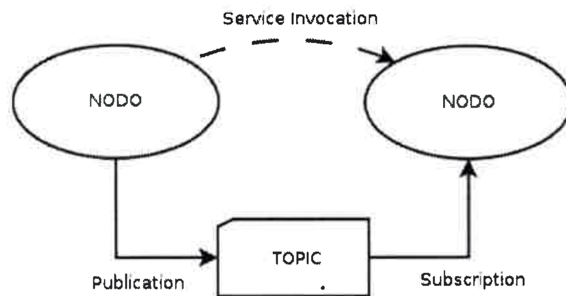


Figura 4.3.1.: Arquitectura general sistema ROS [32]

En el paradigma Publicador-Subscriptor: Un nodo publica datos a un Tópico (canal) y otro se subscribe al mismo para acceder a los datos. Pueden haber varios publicadores y subscriptores. Es un modelo de funcionamiento flexible (asincrónico). Aquí el maestro actúa como un servicio de nombres. Almacena la información de registro de los temas (tópicos) y servicios de los nodos ROS. Los nodos se comunican con el Maestro para reportar su información de registro. A medida que los nodos se comunican con el Maestro, pueden recibir información sobre otros nodos inscriptos y hacer las conexiones, según corresponda. El Maestro también hará devoluciones de llamada a los nodos cuando se ejecutan cambios en la información de registro, así estos pueden crear dinámicamente las conexiones con los nuevos nodos.

Por otro lado, el paradigma Servidor-Cliente permite la funcionalidad Petición-Respuesta. Tanto las peticiones como las respuestas son mensajes ROS y pueden ser estándar (librería "std_msgs") o personalizados. Esto último se logra configurando archivos .msg. Además, el formato de la petición de servicio y la respuesta se pueden configurar en los archivos .srv. Regido por XML-RPC y de comportamiento bloqueante.

Los nodos se conectan a otros nodos directamente, el Maestro sólo proporciona información de búsqueda, al igual que un servidor DNS. Los nodos que se suscriben a un tema solicitarán las conexiones desde los nodos que publican ese tema y se establecerá la conexión mediante un protocolo de conexión. El protocolo más común en ROS se llama TCPROS, que utiliza sockets TCP / IP estándar. La figura 4.3.2 esquematiza la interacción entre el Master y los nodos en un proceso de comunicación ROS.

4.3.3. Nivel de comunidad

Los conceptos de esta sección tienen que ver con el intercambio de conocimientos y softwre entre usuarios/comunidades.

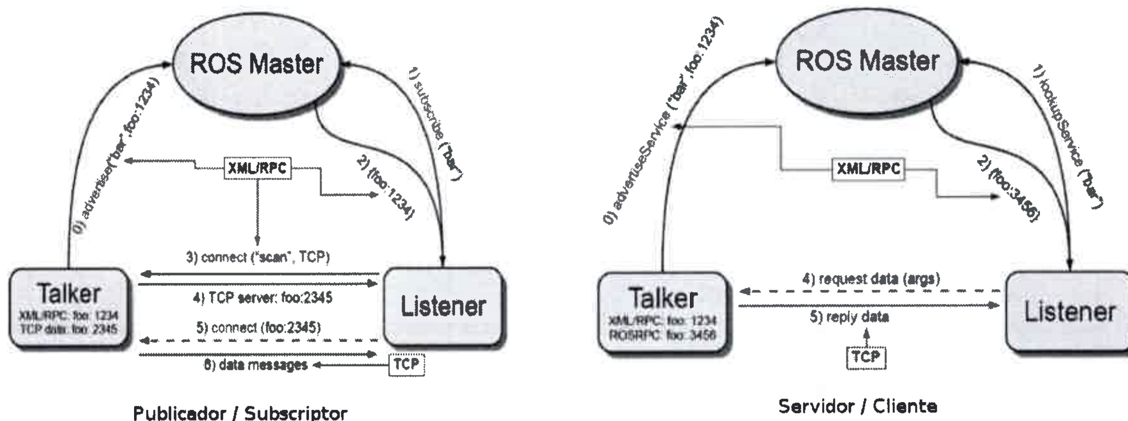


Figura 4.3.2.: Interacción entre Master y nodos.[33][34]

- Distribuciones: Las distribuciones ROS son colecciones de pilas de software que se pueden instalar. Las distribuciones juegan un papel similar al de las distribuciones de Linux: hacen más fácil la instalación de una colección de software, y también mantienen versiones consistentes en un conjunto de software.

- Repositorios: ROS se basa en una red federada de repositorios de código, en los que diferentes instituciones pueden desarrollar y lanzar sus propios componentes de software del robot. Así, se genera un crecimiento colaborativo de la potencialidad de la arquitectura.

- Listas de correo: Las lista ROS de distribución es el principal canal de comunicación sobre los nuevos cambios a ROS, así como un foro para hacer preguntas sobre el software.

- ROS Respuestas: Un sitio para responder a sus preguntas relacionadas con Ros.

- Blog : El blog de Willow Garage ofrece actualizaciones periódicas, incluyendo fotos y videos.

- El ROS Wiki: El ROS comunidad Wiki es el principal foro para la documentación de información sobre ROS. Cualquier persona puede inscribirse para crear una cuenta y contribuir con su propia documentación, proporcionar correcciones o actualizaciones, escribir tutoriales y más.

Resulta entonces evidente que el armado de la documentación de ROS y las posibilidades de tener una experiencia de aprendizaje fluida es muy buena. Además, su filosofía de compartir código entre usuarios permite el crecimiento de la arquitectura de forma natural y adaptada a las necesidades reales de los usuarios.

4.4. Comparación de ROS con otros Middleware para robótica

En este punto del capítulo es conveniente recordar el propósito general de utilizar un middleware; según [24]:

> Un middleware para robótica está diseñado con el fin de administrar la heterogeneidad del hardware, mejorar la aplicación de software calidad, simplificar el diseño de software, y reducir los costos de desarrollo.

Si bien este objetivo general está bien delimitado y es claramente definido, esto no significa que sólo exista una herramienta que pueda cumplirlo. Es más, cuando se investiga, existen variadas arquitecturas de software que se utilizan en robótica. El cuadro 4.1 muestra una comparativa simplificada extraída de [24]

Nombre	Modelo de sistema	Modelo de control	Simul.	S.O.	Abierto	Tecnología	Tiempo Real	Entorno Distri-buido	Seguridad
ROS	Framework basado componen-tes; Publicador-Subscriptor	Orientado a mensajes	Sí	Win-Linux	Sí	Orientado a mensajes; Servicios RPC	Sí	Sí	No
Pyro	Independientes de la arquitectura	-	Sí	Win-Linux	Sí	Basado en sockets TCP; XML; HTTP; SOAP; OpenGL	No	No	Sí
Roboframe	Orientado a mensajes; publicador-Subscriptor y mecanismos de memoria compartida	Basado en mensajes	Sí	Win-Linux	No	Basado en sockets	No	Sí	No
OROCOS	Librerías C++; OCL; KDL; BFL	Basado en control	No (pa-quete Simu-link)	Win-Linux	Sí	ACE/TAO, CORBA	Sí	No	No

Cuadro 4.4.1.: Comparativa entre algunos frameworks para robótica

En el cuadro se puede apreciar que un punto particular de ROS es que no cuenta con un

UNIVERSIDAD NACIONAL DE RIO CUARTO
BIBLIOTECA

sistema interno de seguridad en las conexiones. Sin embargo, esto no es una debilidad insalvable dado que siempre está la posibilidad de utilizar conexiones seguras SSH y de hecho en la web oficial de ROS se halla un tutorial paso a paso de como hacer la configuración.

Además, puede verse que el sistema permite trabajar en más de un sistema operativo, lo cual es importante porque en el ámbito académico suelen utilizarse tanto sistemas basdos en UNIX como también propietarios. Es interesante agregar que ROS es un sistema Open Source, lo que significa que aparte de ser de obtención gratuita, el código fuente es asequible a cualquiera que desee mejorarlo, estudiarlo o adaptarlo.

Por otro lado, la programación de ROS le permite trabajar con sistemas distribuidos e incluso puede utilizarse en sistemas de tiempo real. Sin embargo, si lo que se necesita es controlar un sistema de tiempo real de misión crítica[1] lo mejor es OROCOS, dado que es el objetivo de su diseño, mientras que ROS fue pensado como un sistema más genérico. En [35] se puede hallar una discusión interesante sobre el tema. Como reconoce la wiki oficial: "El publicador 'normal' de ROS no es seguro (confiable) en operaciones de tiempo real y por tanto no debería ser usado en el lazo de actualización de un controlador en tiempo real". Por tanto, aquí se encuentra otro talón de Aquiles de ROS.

Como principales ventajas de ROS se pueden mencionar [36]:

- Comunicación entre procesos fácil de usar y bastante versátil.
- Permite facilmente la integración de un gran número de herramientas, entre ellas las orientadas a visualización robótica, sensores de datos, algoritmos de percepción así como también drivers de bajo nivel para sensores de uso común.
- Contiene herramientas de manejo y supervición de los mensajes.

¿Cuán complejo es la puesta en marcha del núcleo (core) de ROS? Tras la instalación del metasistema operativo siguiendo las instrucciones de la página oficial, con sencillamente escribir en una consola el comando "roscore" se tiene una respuesta como la mostrada en la figura 4.4.1 y se consigue que el controlador o cerebro de la arquietectura ROS ya esté listo para coordinar las operaciones y comunicaciones entre los nodos de comunicación (publisher y subscriber).

4.5. Rosbridge: Un "puente" entre el ROS y la Web

Entre las herramientas con que cuenta ROS es posible mencionar una que resulta muy interesante a la hora de diseñar la interfaz que permita al usuario teleoperar fácilmente e

[1] Se define un sistema de tiempo real de misión crítica a aquellos procesos en que el fallo al cumplir una restricción temporal tiene consecuencias severas (Wikipedia).

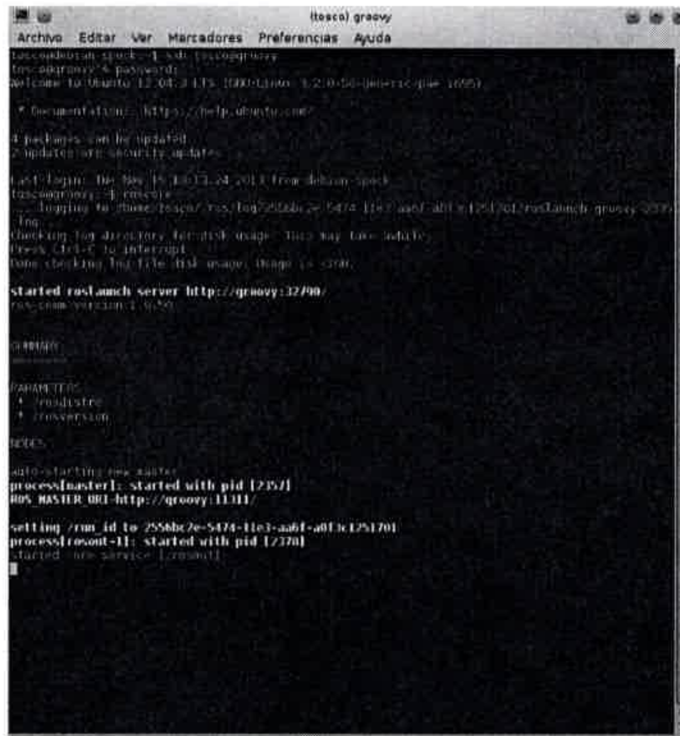


Figura 4.4.1.: Ejemplo del “core” de ROS lanzado en una PC llamada “Groovy”

incluso desde un teléfono o una tablet con acceso a internet un laboratorio remoto: la suite Rosbridge.

Según la wiki oficial : “La suite Rosbridge provee una API JSON[2] hacia funcionalidades ROS para programas que no forman parte de dicha arquitectura”. Entonces puede decirse que Rosbridge es una suerte de “puente” entre ROS y, por ejemplo, el explorador de internet (el cual tiene generalmente soporte para Javascript[3]), el cual puede ser usado como interfaz moderna, y ágil para que un usuario (por ejemplo un estudiante) teleopere un laboratorio remoto en el marco de una experiencia de laboratorio. Es digno de notar que la nueva versión del estandar HTML (el lenguaje básico de Internet), HTML 5, en conjunto con CSS3 (la revisión 3 del lenguaje utilizado para describir el aspecto y formato de las paginas de internet) posibilitan la presentación de interfaces web vistosas, ágiles y adaptables a dispositivos móviles (teléfonos y tabletas electrónicas).

Rosbridge se constituye de dos partes fundamentales: el protocolo y la implementación

[2]JSON (JavaScript Object Notation) es un formato ligero de intercambio de datos muy útil cuando el tamaño del flujo de datos entre cliente y servidor es de vital importancia y la fuente es de fiar. Es muy usado por Google y Yahoo. Puede usarse en conjunto en la misma aplicación con XML. (Wikipedia)

[3]JavaScript es un lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos,3 basado en prototipos, imperativo, débilmente tipado y dinámico. Se utiliza principalmente en su forma del lado del cliente (client-side), implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas aunque existe una forma de JavaScript del lado del servidor (Server-side JavaScript o SSJS). Su uso en aplicaciones externas a la web, por ejemplo en documentos PDF, aplicaciones de escritorio (mayoritariamente widgets) es también significativo. (Wikipedia)

del mismo.

Con respecto al primero cabe decir que está basado en JSON y que en teoría cualquier lenguaje que maneje dicha notación podrá comunicarse con ROS mediante el protocolo Rosbridge. Un ejemplo de dicho protocolo (implementado en javascript), por ejemplo para suscribirse a un tópico (canal) es el siguiente:

```
var listenersp = new ROSLIB.Topic({
ros : ros,
name : '/sp',
messageType : 'std_msgs/Float64'
});
listenersp.subscribe(function(message) {
var spdetec=message.data;
document.getElementById('sp_text').innerHTML=spdetec.toFixed(3);
});
```

Puede verse en el ejemplo que listenersp es un objeto que representa el tópico de interés y sobre el cual se ejecuta el método "subscribe" para obtener el dato y representarlo en un campo de formulario llamado "sp_text". Más allá de los detalles de javascript en la definición del objeto "listenersp" se ve claramente la simpleza de la declaración JSON en la cual no solo se menciona que la comunicación es con ROS, sino que el tópico a conectarse es "/sp" y que el tipo de dato a recibir es el tipo flotante (64 bits de precisión).

Para poner en marcha el servidor Rosbridge, tras su instalación siguiendo las indicaciones de la página oficial, basta con correr en una terminal: rosrun rosbridge_server rosbridge_websocket . En la figura 4.5.1 se ve una captura de pantalla alusiva.

Con respecto alpaquete rosbridge_suite puede decirse que es el encargado de implementar el protocolo antes descripto y proveer una capa de transporte tipo websocket (nótese la expresión "rosbridge_websocket" en la sentencia rosrun)

Se entiende por websocket a la tecnología proporciona un canal de comunicación bidireccional y full-duplex sobre un único socket TCP. Está diseñada para ser implementada en navegadores y servidores web, pero puede utilizarse por cualquier aplicación cliente/servidor. [16]. Como las conexiones TCP ordinarias sobre puertos diferentes al 80 son habitualmente bloqueadas por los administradores de redes, el uso de esta tecnología proporciona una solución a este tipo de limitaciones proveyendo una funcionalidad similar a la apertura de varias conexiones en distintos puertos, pero multiplexando diferentes servicios WebSocket sobre un único puerto TCP (a costa de una pequeña sobrecarga del protocolo). Del lado del cliente, esta tecnología ya está implementada en la mayoría de los navegadores principales.

Con lo antes mencionado resulta evidente porqué el rosbridge es una herramienta importante en el marco de esta tesis. Dado que los laboratorios remotos están generalmente

Figura 4.5.1.: Ejemplo de Rosbridge lanzado en una PC llamada "Groovy" y esperando conexiones

instalados dentro de los campus universitarios, es normal que las conexiones a internet disponibles estén segurizadas con políticas restrictivas en lo que a apertura de puerto se refiere. Así, la tecnología websocket permite que con la instalación de un servidor web en el campus y la consiguiente apertura del puerto estándar 80 ya sea posible desde fuera de las instalaciones conectarse al sistema ROS y realizar la práctica experimental remota.

4.6. MJPG-streamer: Útil herramienta para la visualización de streaming de video

Es interesante que en las experiencias remotas de teleoperación siempre se necesita la inspección visual. Ahora bien, surge la pregunta: ¿cómo añadir esta prestación al sistema sin afectar la arquitectura ROS? La respuesta radica en pensar en un servidor que tome el flujo de video y lo inyecte en la red en forma de streamer independiente para que desde la web pueda verse el estado de situación de la planta. Una aplicación que permite realizar esto de forma sencilla es Mjpg-Streamer.

Dicha aplicación corre en modo consola y sirve para difundir vía una red IP fotogramas con compresión JPEG desde una webcam hasta un visualizador como por ejemplo un explorador web, VLC o cualquier dispositivo móvil con navegador web [37].

El comando que pone en funcionamiento este servidor de streaming de video puede ser

como sigue:

```
mjpg_streamer -i "input_uvc.so -d /dev/video0 -r 160x120" -o "output_http.so -p 8080
-w /usr/local/www" &
```

Puede verse que mjpeg_streamer funciona como un pegamento entre "plug-ins" que realizan distintas tareas. Por ejemplo, arriba puede apreciarse que tras indicar el parámetro "-i" (entrada) se invoca el plug-in "input_uvc.so" (que sirve para conectar con cámaras compatibles con V4L2) con indicación de la ubicación de la cámara en el sistema linux y una definición del tamaño de fotograma. Además, se indica tras el parámetro "-o" (salida) el plug-in "output_http.so" (es un webserver http totalmente funcional) con indicación de usar el puerto 8080 y servir la web en la carpeta /usr/local/www. El signo "&" es para que la terminal no se bloquee y lance el proceso por separado.

En la figura 4.6.1 puede verse una captura de video desde una webcam y visualizada desde el navegador Iceweasel.

Figura 4.6.1.: Captura de streaming de video generado desde una webcam y transmitido mediante MJPG-Streamer

¿Conviene utilizar MJPG-Streamer? Esta herramienta *no depende* de ROS y por tanto, una ventaja al utilizarla es que una falla en la emisión de video no afecta al sistema del laborio remoto en sí. Además, para tamaños pequeños (por ejemñplo 160x120) la carga en la red no es significativa para una conexión a Internet estándar. También, hay que reconocer que la compatibilidad de los navegadores para mostrar MJPEG streamer en HTML5 es mucho mayor que si se usara un flujo de video comprimido con otros estándares (Por ejemplo H.264). También conviene destacar que MJPG-Streamer es muy fácil de manejar e instalar en sistemas linux como los utilizados para este trabajo de tesis e incluso, sin ser parte de ROS, es compatible con éste en algunos aspectos (por ejemplo al poder lanzarse ambos desde un sólo archivo de configuración). Es menester mencionar que esta herramienta fue diseñada para sistemas con pocas prestaciones de CPU y RAM; sirva de boton de muestra la captura de la figura 4.6.1, la cual muestra el flujo que procede de un servidor MJPG-Streamer instalado en una placa BeagleBone rev.5 con S.O. Ubuntu, la cual posee 256

MB de Ram un procesador de 700 MHz. Como desventaja puede mencionarse que para grandes tamaños de fotogramas el requerimiento de ancho de banda podría ser un escollo.

4.7. Conclusiones

El presente capítulo aportó información sobre el sistema MOM escogido: ROS. Pudieron apreciarse algunos detalles de su arquitectura interna y su filosofía de funcionamiento. Además, se mostraron algunas herramientas dignas de destacar, como el RosBridge y el MJPG-Streamer y se fue explicando cómo serían utilizadas cada una de ellas en el contexto de la implementación del laboratorio remoto que sirve de caso de estudio en la presente tesis. También se adjuntaron capturas de pantallas alusivas y se mostraron algunos comandos de uso básico a modo introductorio. En el Anexo-C se pueden hallar algunos ejemplos más.

Por otro lado, resultó de interés la comparación entre plataformas de software realizada, la cual ha discriminado las virtudes y algunos puntos débies de la plataforma escogida. Entre las virtudes se destacan: orientado a mensajes, multisistema operativo, código libre, permite trabajar en entornos distribuidos, posee simulador y está bien documentado. Entre los puntos débiles se tiene que no posee seguridad implementada (puede salvarse este escollo utilizando conexiones seguras "ssh") y que no es el middleware ideal para aplicaciones exigentes en tiempo real (se vio que OROCOS está diseñado con este fin).

También, a lo largo del capítulo se dejó claro los aspectos que se tuvieron en cuenta a la hora de decantarse por ROS dentro del abanico de posibilidades de los MOMs. Entre dichas virtudes se recalcan: sus características de rendimiento y funcionamiento, su capacidad de trabajar con lenguaje python y la buena documentación disponible junto a la abundante cantidad de módulos, ejemplos didácticos y paquetes (incluso pilas de software) compartidos.

Habiendo explicado las cuestiones relacionadas con software, se procede a continuación a explicar el papel de los sistemas embebidos como nodos de adquisición en el contexto de los laboratorios remotos. Posteriormente se hará un análisis del caso de estudio construído como parte del presente trabajo de maestría y finalmente se hará un análisis y sendos comentarios de los resultados.

CAPÍTULO V USO DE SISTEMAS EMBEBIDOS COMO NODOS DE ADQUISICIÓN Y PROCESAMIENTO

Con el avance de la microelectrónica, se han empezado a utilizar dispositivos diseñados para propósitos específicos, con moderadas prestaciones y a bajo costo llamados: sistemas embebidos o empotrados. Dichos dispositivos no suelen tener el aspecto que uno esperaría de un ordenador y constructivamente se caracterizan por tener empotrados en su placa base la mayoría de los componentes, por ejemplo: placa de video, audio, modem, etc.

En virtud de que los costos son un factor determinante en el diseño de los laboratorios remotos, el uso de sistemas embebidos representa una opción interesante a tener en cuenta. Por eso es que a continuación se hace una caracterización general de los sistemas empotrados para luego describir la tecnología utilizada en el desarrollo del caso de estudio generado para la presente tesis dentro del contexto de la arquitectura ROS (descrita en el capítulo anterior).

5.1. Caracterización de los sistemas embebidos

Una definición concisa de sistema embebido o empotrado podría ser la siguiente [16]:

> Un sistema embebido (anglicismo "embedded") o empotrado es un sistema de computación diseñado para realizar una o algunas pocas funciones dedicadas, frecuentemente en un sistema de computación en tiempo real.

Puede decirse entonces que es un sistema electrónico que está contenido ("embebido") dentro de un equipo completo que incluye, por ejemplo, partes mecánicas y electromecánicas [38]. Como una aplicación derivada se puede hablar de "Linux embebido", el cual se refiere al uso del núcleo Linux en un sistema embebido, como por ejemplo PDA, teléfonos móviles, robots, enrutadores / servidores, dispositivos electrónicos y aplicaciones industriales con microcontroladores y microprocesadores. Si bien existen otros sistemas operativos empotrados, el linux presenta ventajas significativas por ocupar un tamaño reducido y por ser de código libre con amplia utilización en el campo.

Típicamente, el corazón de este tipo de sistemas es un microcontrolador, aunque los datos también pueden ser procesados por un DSP, una FPGA, un microprocesador o un ASIC,

y su diseño está optimizado para reducir su tamaño y su costo, aumentar su confiabilidad y mejorar su desempeño. El diseño de sistemas embebidos es un motor clave de la industria y del desarrollo tecnológico, y es un campo que en los últimos años ha crecido notablemente en la Argentina [38].

Un desarrollo en el que se focaliza la atención del presente trabajo son las computadoras embebidas o microcomputadoras. A continuación se hace una breve reseña sobre las mismas para después pasar a mostrar una aplicación de interés sobre ellas aplicada al tratamiento digital de imágenes.

5.2. Arquitectura básica

Una computadora embebida posee una arquitectura semejante a la de una de escritorio. Se describen en el cuadro 5.1 algunas de las partes fundamentales [16, 39].

Parte	Descripción sucinta
Procesador	Es el encargado de realizar las operaciones de cálculo principales del sistema. Ejecuta código para realizar una determinada tarea y dirige el funcionamiento de los demás elementos que le rodean. Existe una decisión de compromiso entre costo y capacidad de cálculo al dimensionar este elemento.
Memoria	En ella se encuentra almacenado el código de los programas que el sistema puede ejecutar así como los datos. La memoria RAM es volátil y su principal limitante es la velocidad de lecto-escritura.
Caché	Memoria más rápida que la principal en la que se almacenan los datos y el código accedido últimamente. Permite acelerar búsquedas repetitivas en las microtareas del sistemas.
Perisféricos	Permiten la comunicación desde y hacia el exterior del sistema embebido. Las pricipales tipos de perisféricos incluyen salidas binarias, seriales, analógicas, display (monitores, pantallas) y basadas en tiempo (ej.: contadores)
Software	Permite al sistema desempeñar tareas y le da valor agregado al sistema.
Algoritmos	Son la parte clave del software y definen la forma en que se resuelven las tereas. Se basan en muchas veces modelos matemáticos y procedimientos lógicos. Existen estándares.

Cuadro 5.2.1.: Partes fundamentales de un ordenador empotrado

5.3. Aplicaciones de los sistemas embebidos

Se utilizan en aplicaciones varias y de naturaleza distinta. Por ejemplo, en las plantas de fabricación pueden controlar procesos de montaje o fabricación. En los puntos de venta

coordinan dispositivos de entrada y salida para asegurar operación continua. En el ámbito del turismo, son cada vez más comunes los puntos de información para respuestas personalizadas a través de una pantalla táctil y una interfáz amigable e interactiva.

Por otro lado, en lo que a televisión satelital y por cable se refiere, encontramos sistemas embebidos en los set-up boxes y decodificadores que se encargan de tareas relativas a la señal de video, codificación y software para la navegación por internet.

La aviónica no es la excepción; numerosos sistemas de radar en este campo utilizan los sistemas empotrados para coordinar las señales y ubicar precisamente el avión, sus obstáculos y permitir la maniobrabilidad en condiciones difíciles de vuelo.

Además, se puede incluir a los cajeros automáticos, los equipos médicos de hospitales, las ambulancias bien equipadas, las pasarelas (gateway) de Internet, las máquinas de revelado automático de fotografías.... en fin son numerosas y variadas las aplicaciones de los sistemas embebidos tanto en el presente como por descubrir en el futuro.

¿Qué ventajas trae aparejado el utilizar este tipo de sistemas? Por un lado, la especificidad de la aplicación o sistema operativo. Sí, muchas veces el programa se realiza en el ensamblador propio del microprocesador, con lo cual se evitan redundancias de código o capacidades ociosas por ser una implementación ad-hoc. Además, la reducción en costos de hardware es significativa.

También, si se utiliza un sistema operativo (S.O.) embebido ello permite utilizar la complejidad de los programas diseñados para PCs de escritorio pero con la capacidad y tamaños mínimos de los ordenadores embebidos. También, dicho tipo de S.O. permite utilizar herramientas de desarrollo de software conocidas y dominadas por muchos programadores en el mundo.

Huelga decir que el consumo en potencia eléctrica es, en general, mínimo y los tamaños de dispositivo son pequeños.

Entonces, tras el análisis llevado a cabo, puede notarse que la utilización de computadores embebidos es una opción atrayente en el diseño de laboratorios remotos.

5.4. Beaglebone: un microcomputador de buenas prestaciones

Beagle Bone es un ordenador en pequeño pero con el que también es posible controlar hardware externo a través de sus pines. Su procesador es un ARM cortex A8 que funciona a unos 700mhz (más adelante se precisan más datos) y cuenta con 256MB de RAM. Con estas prestaciones es posible ejecutar un sistema operativo como el existente en un ordenador, siempre y cuando exista una versión compilada para procesador ARM; existen distribuciones de linux para arm. Se utilizó para las pruebas realizadas Linux Ubuntu 12.04.3 LTS (GNU/Linux 3.8.13-bone27 armv7l).

Es realmente pequeño, unos 9cm de largo y 5,5 de ancho y cuenta con un lector de trajetas microsd (que hace las veces de "disco duro" para almacenar el SO y los datos), un host USB (para conectar dispositivos), un conector RJ45 y un micro USB para vincularlo al PC.

En la figura 5.4.1 se muestra una vista de una placa beaglebone etiquetada para señalar las partes más importantes.

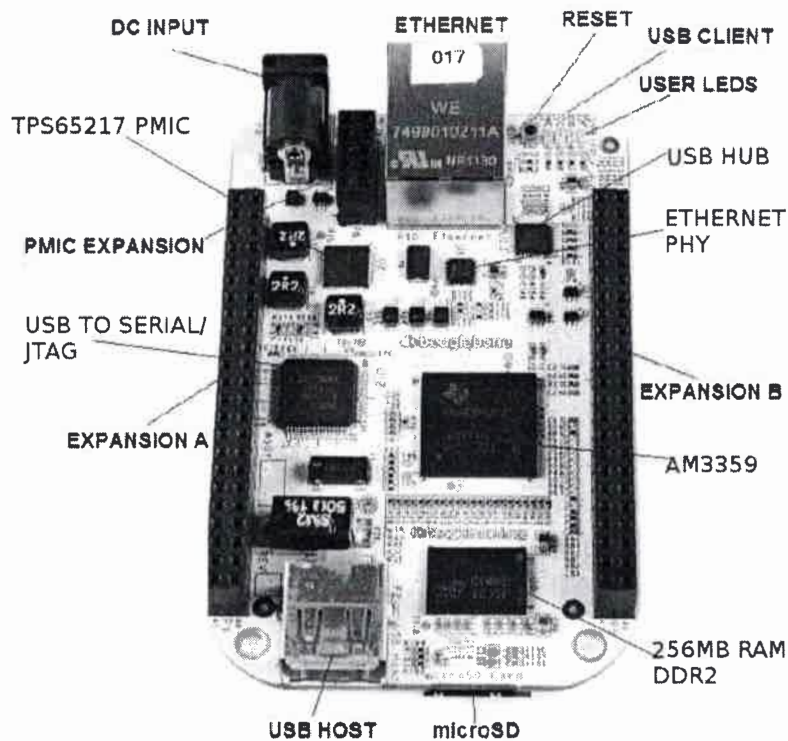


Figura 5.4.1.: Vista de un beaglebone

Como apreciarse, el microprocesador es un AM3359. Su estructura interna se muestra en el diagrama en bloques de la figura 5.4.2. Dicho procesador tiene una frecuencia variable según la tensión de alimentación. Así, cuando es alimentado por el USB su velocidad de reloj es 500MHZ, mientras que si se la alimenta desde la entrada DC sube a 700MHZ.

Con respecto a su memoria RAM es de 256 MB DDR2 de 400MHZ. Es suficiente para correr un sistema operativo modo consola de manera fluida. No tiene salida de video.

El regulador de tensión es un PMIC TPS65217B. Posee alimentación vía USB (conector miniUSB) ó conector 5VDC externo. EL PCB es de 6 capas y tiene visibles indicadores de alimentación y 4 leds controlables por el usuario (por defecto están asignado a diversas funciones del S.O.) También la placa cuenta con un conector RJ45 para interfaz ethernet 10/100 Mbps. Esta placa tiene también un botón de RESET (reinicio) y una protección de apagado por sobretensión fijada en 5.6V MAX.

Entre los parámetros de interés está el consumo. La figura 5.4.3 muestra una tabla intere-

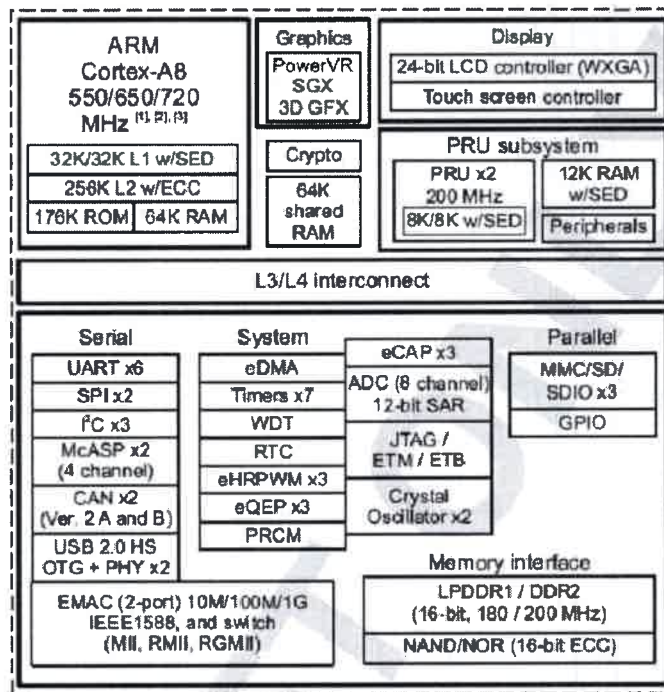


Figura 5.4.2.: Diagrama en bloques del AM3359

sante al respecto.

MODE	USB	DC	DC+USB
Reset	**180**	**60**	**190**
UBoot	**363**	**230**	**340**
Kernel Booting (Peak)	**502**	**350**	**470**
Kernel Idling	**305**	**170**	**290**

Figura 5.4.3.: Tabla de consumo del Beaglebone

Puede notarse que la utilización del usb requiere más electrónica funcionando y por tanto demanda unos 120mA más de corriente eléctrica. En el peor de los casos el pico de consumo ronda el medio Ampere. Sin embargo, si se le va a acoplar dispositivos al puerto USB se recomienda el uso de una fuente externa de 2A.

¿Por qué la elección de la placa BeagleBone? En resumidas cuentas puede decirse que se la eligió por cinco razones:

- Cumplía con los requerimientos mínimos que se necesitaban para el proyecto.
- Era una minicomputadora con bancos de expansión y no sólo un microcontrolador con interfaces (Ej. Arduino)
- El costo era accesible.
- Se conseguía en el país y en tiempos razonables.
- La documentación es muy buena y hay mucha gente trabajando en esto.

5.4.1. Bancos de expansión

Uno de los aspectos más importantes que se necesitaba para el proyecto enmarcado en esta tesis era una placa de desarrollo que tuviera pines de expansión con prestaciones útiles y variadas. Este modelo de microcomputador posee 2 bancos de expansión (denominados P8 y P9). En las figuras 5.4.4 y 5.4.5 se muestran las funciones de los pines. En la web se halla el manual de referencia del BeagleBone para más información.

SIGNAL NAME	PROC	CONN		PROC	SIGNAL NAME
	GND	1	2	GND	
GPIO1_6	R9	3	4	T9	GPIO1_7
GPIO1_2	R8	5	6	T8	GPIO1_3
TIMER4	R7	7	8	T7	TIMER7
TIMER5	T6	9	10	U6	TIMER6
GPIO1_13	R12	11	12	T12	GPIO1_12
EHRPWM2B	T10	13	14	T11	GPIO0_26
GPIO1_15	U13	15	16	V13	GPIO1_14
GPIO0_27	U12	17	18	V12	GPIO2_1
EHRPWM2A	U10	19	20	V9	GPIO1_31
GPIO1_30	U9	21	22	V8	GPIO1_5
GPIO1_4	U8	23	24	V7	GPIO1_1
GPIO1_0	U7	25	26	V6	GPIO1_29
GPIO2_22	U5	27	28	V5	GPIO2_24
GPIO2_23	R5	29	30	R6	GPIO2_25
UART5_CTSN	V4	31	32	T5	UART5_RTSN
UART4_RTSN	V3	33	34	U4	UART3_RTSN
UART4_CTSN	V2	35	36	U3	UART3_CTSN
UART5_TXD	U1	37	38	U2	UART5_RXD
GPIO2_12	T3	39	40	T4	GPIO2_13
GPIO2_10	T1	41	42	T2	GPIO2_11
GPIO2_8	R3	43	44	R4	GPIO2_9
GPIO2_6	R1	45	46	R2	GPIO2_7

Figura 5.4.4.: Asignaciones de expansión P8 (Beaglebone)

Entre las funciones a las que se accede a través de estos pines se cuentan:

- Puertos de entrada/salida digital (@3.3V)
- Entradas analógicas (@1.8V MAX)
- Salidas PWM.
- Puerto Serie
- Temporizadores (Timers)
- Acceso a GMPC, MMC1.
- Puertos SPI y I2C.

Un detalle interesante es que esta placa cuenta con 7 conversores A/D a 100K muestras por segundo.

Para el desarrollo del caso de estudio pensado en el presente trabajo resultó necesario utilizar varias de estas capacidades. Como se verá en un capítulo posterior, resultó todo

UNIVERSIDAD NACIONAL DE ROSARIO
BIBLIOTECA

SIGNAL NAME	PIN	CONN		PIN	SIGNAL NAME
	GND	1	2	GND	
	VDD_3V3EXP	3	4	VDD_3V3EXP	
	VDD_5V	5	6	VDD_5V	
	SYS_5V	7	8	SYS_5V	
PWR_BUT*		9	10	A10	SYS_RESETn
UART4_RXD	T17	11	12	U18	GPIO1_28
UART4_TXD	U17	13	14	U14	EHRPWM1A
GPIO1_16	R13	15	16	T14	EHRPWM1B
I2C1_SCL	A16	17	18	B16	I2C1_SDA
I2C2_SCL	D17	19	20	D18	I2C2_SDA
UART2_TXD	B17	21	22	A17	UART2_RXD
GPIO1_17	V14	23	24	D15	UART1_TXD
GPIO3_21	A14	25	26	D16	UART1_RXD
GPIO3_19	C13	27	28	C12	SPI1_CS0
SPI1_D0	B13	29	30	D12	SPI1_D1
SPI1_SCLK	A13	31	32	VDD_ADC(1.8V)	
AIN4	C8	33	34	GNDA_ADC	
AIN6	A5	35	36	A5	AIN5
AIN2	B7	37	38	A7	AIN3
AIN0	B6	39	40	C7	AIN1
CLKOUT2	D14	41	42	C18	GPIO0_7
	GND	43	44	GND	
	GND	45	46	GND	

*PWR_BUT is a 5V level as pulled up internally by the TPS65217B. It is activated by pulling the signal to GND.

Figura 5.4.5.: Asignaciones de expansión P9 (Beaglebone)

un desafío encontrar un kernel de linux que accediera sin problemas al hardware de la placa y lo comandara con cierta confiabilidad.

5.4.2. Primera conexión con BeagleBone

Para conectarse por primera vez a este dispositivo, es menester usar la conexión usb y utilizar el comando:

```
screen /dev/ttyUSB0 115200
```

Tras iniciar sesion (por defecto: usuario *ubuntu*, contraseña *temppwd*) es posible configurar la red mediante consola en el sistema linux y ya es posible acceder vía *ssh* desde otro PC. Para más información sobre cómo realizar esto y cómo cargar un sistema operativo distinto al que trae por defecto (Linux Amstrong) es posible remitirse a sendos tutoriales en Internet.

5.5. Uso de la BeagleBone como nodo ROS

Como se explicó en el capítulo anterior, un nodo ros es el encargado de realizar las tareas de cálculo. En otras palabras, es un ejecutable que realiza las operaciones de interés

dentro del sistema ROS. Además, también quedó establecida la característica fuertemente modular de la arquitectura ROS. Eso significa en la práctica que al desarrollar un proyecto dado se pensará en subdividir las tareas en otras más pequeñas, interdependientes y distribuibles en distintas unidades de cómputo. Cada uno de esos módulos puede constituir un nodo. ¿Será posible correr un nodo en la placa BeagleBone? La respuesta es un rotundo sí.

5.5.1. Primeros pasos para crear un paquete donde correrá el nodo

Siguiendo con la guia de la web oficial de ROS es posible instalar (los paquetes compilados para arquitectura ARM) y configurar el espacio de trabajo en el BeagleBone(BB). Una vez hecho esto es posible crear un paquete con el comando[1]:

```
roscreate-pkg [package_name] [depend1] [depend2] [depend3]
```

Con dicha sentencia se consigue crear un paquete de software con un nombre propio y se pueden configurar las dependencias (otros paquetes previamente existentes), siendo las más comunes: *std_msgs*, *rospy*, *roscpp*.

Para que el sistema ROS detecte nuestro paquete se procede a ejecutar: "rospack profile". Tras hacerlo, desde cualquier posición en el sistema de archivos será posible ir a la carpeta principal del paquete creado con el comando:

```
roscd [package_name]
```

También se procede a compilar el paquete (se requerirá cumplir con todas las dependencias) con el comando:

```
rosmake [package]
```

ROS posee varios comandos a modo consola que pueden utilizarse para facilitar el trabajo con esta arquitectura de software; estos se pueden encontrar bien documentados en [40]

Una estructura de paquete típica se muestra en la figura 5.5.1.

Como puede apreciarse, hay un archivo que constituye el manifiesto del paquete (manifest.xml). En la figura 5.5.2 se ve el contenido del mismo.

Resulta evidente la información disponible en este archivo (de notación XML): Descripción (acá solo tiene el nombre del paquete, pero podría ser más amplia la información), el autor, la licencia, información sobre documentación y revisión. Además, se pueden visualizar claramente las dependencias del paquete rc_sim, las cuales son las anteriormente comentadas: std_msgs, rospy y roscpp.

[1]Como se dijo en la sección 4.3 se trabajará con el sistema ROSBUILD.

Figura 5.5.1.: Ejemplo de carpeta de un paquete llamado rc_sim creado en un BB

Figura 5.5.2.: Contenido del manifiesto de un paquete llamado rc_sim creado en un BB

Por otro lado se ve dos carpetas de interés: "src" y "launch" (esta fue creada por el usuario). La primera es la que contiene todos los archivos de código ejecutables (deben tener permiso de ejecución en el S.O.) programados en el lenguaje que se desee, y la segunda carpeta contiene el archivo de lanzamiento ROS de extención ".launch". Cabe aclarar que este archivo tiene estructura XML y sirve para definir parámetros, configuraciones, lanzar procesos y nodos, todo considerando un sistema distribuido, desde un sólo archivo.

5.5.2. Un nodo ejemplo

Tomando de referencia el paquete rc_sim anteriormente mencionado, es posible tomar a modo de ejemplo un nodo: nodo_planta. Este nodo está programado en el archivo nodo_planta.py (el cual tiene permiso de ejecución) y simula una planta muy sencilla de interés para este proyecto, que simula el llenado de un tanque de agua (más información en el capítulo 7). A continuación se muestra en la figura 5.5.3 la porción del código que resulta de interés en el uso de ROS (<...> indica que ahí va código que no es pertinente y por tanto no se muestra).

Puede notarse en la antedicha figura que, tras el comentario inicial (rótulo) se importan las librerías de código a utilizar. También, al comienzo, se carga el manifiesto del paquete que contiene a "nodo_planta" que es "rc_sim". Además, en las líneas subsiguientes, se importa la librería cliente rospy (que permite usar ROS dentro de Python) y después (línea

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  """
4      nodo_planta.py - Nodo que simula planta (Circuito RC)
5
6      Copyright (C) 2013 SJTosco
7
8      <...>
9
10 """
11
12 import roslib; roslib.load_manifest('rc_sim')
13 import rospy
14 from std_msgs.msg import Float64, String
15 import <...>
16
17 class Planta:
18     def __init__(self):
19         rospy.init_node('nodo_planta', anonymous=True)
20
21         rospy.on_shutdown(self.cleanup)
22
23         <...>
24         # We don't have to run the script very fast
25         self.rate = rospy.get_param("rr", 5)
26         r = rospy.Rate(self.rate)
27
28         # Publish Yout message to the y topic
29         self.pub_y = rospy.Publisher('y',Float64) #Publicador y
30
31         # Subscribe to the u topic to receive pwm commands.
32         rospy.Subscriber('u', Float64, self.callback_u) #Subscriptor u
33         <...>
34
35         # We have to keep publishing the cmd_vel message if we want the robot to keep moving
36         while not rospy.is_shutdown():
37             <...>
38             self.pub_y.publish(Float64(Y_Cap))
39             r.sleep()
40
41     <...>
42
43     def callback_u(self, data):
44         <...>
45
46
47 if __name__=="__main__":
48     try:
49         Planta()
50         rospy.spin()
51     except rospy.ROSInterruptException:
52         rospy.loginfo("Planta (rc_sim) terminated!")
```

Figura 5.5.3.: Sección del código de nodo_planta.py

14) se importan los tipos de mensajes estandar Float64 y String.

Para quienes conocen la Programación Orientada a Objetos (POO) resulta evidente que lo que se define a continuación es una clase llamada Planta la cual tiene varios métodos, el primero de los cuales es "__init__(self)" que se ejecuta al instanciarse la clase. Reviste interés especial la línea 19 donde se inicializa (o crea) el nodo llamado "nodo_planta" con la opción "anonymous=True" que implica que corra los nodos sin tomar en cuenta los nombres.

A continuación hay varias configuraciones invocando la funciones de "rospy" y después, en las líneas 29 y 32 se crean los objetos publicadores y subscriptores, respectivamente. Puede notarse que ambos manejan un tipo de dato Float64 (punto flotante, alta precisión) y en el caso del subscriptor puede verse que se define el nombre de la función que se ejecutará al recibir un dato desde el tópico (tema) al que se está inscripto, llamada "callback_u". También resulta evidente que hay dos tópicos distintos que resultan de interés: "y" para el publicador y "u" para el subscriptor. Más adelante, en el capítulo 7 se entenderán las razones de esto.

Por otro lado, en la línea 38 puede verse cómo se invoca el método "publish" del objeto publicador para enviar un dato hacia el tópico de interés.

Finalmente, desde la línea 48 puede notarse la intanciación de la clase "Planta" y el comando "rospy.spin()" el cual pone en marcha un ciclo especial en ROS que implica quedar a la espera de que el servicio o nodo deje de estan activo (se apague).

Tras la breve descripción que se hizo del código no se pretende que el lector maneje

ROS de forma fluída, ni mucho menos. En realidad el objetivo es poner en evidencia la simpleza con la que se maneja un publicador y un subscriptor en python e incluso sobre un sistema embebido (el BB). Huelga decir que el proceso sería idéntico en un ordenador de escritorio, con la única diferencia que los paquetes de instalación de la plataforma ROS serían los compilados para una arquitectura compatible (por ej.: x86)

5.5.3. Un nodo servidor MJPEG

En el BB es posible instalar el servidor MJPG-Streamer del cual se ha hablado en la sección 4.6. Una vez instalado, es posible crear un nodo ROS llamado "nodo_camara" contenido en el archivo "nodo_camara.sh" (el cual tiene los permisos de ejecución en el S.O.). Este archivo posee una configuración determinada del servidor de video y su contenido se muestra en la figura 5.5.4.

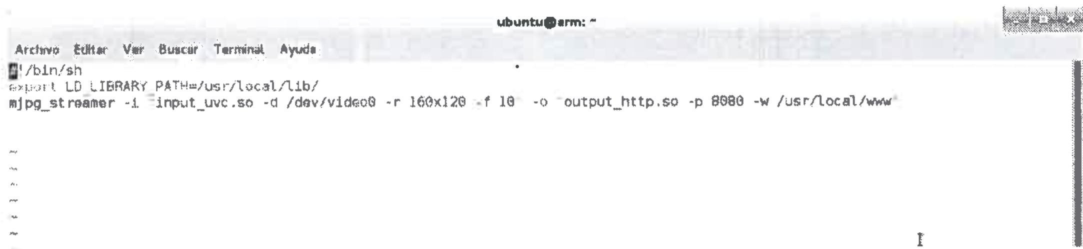


Figura 5.5.4.: Sección del código de nodo_camara.sh

Puede apreciarse una configuración similar a la explicada en la sección 4.6 y el resultado que produe es el mismo de la figura 4.6.1 que ahora (por comodidad) se repite en la figura 5.5.5 sólo que enmarcado en una página web.

Para correr a andar el nodo ROS "nodo_camara" se invocó el archivo lanzador (pru.launch) creado con dicho propósito y ubicado en la carpeta "launch" dentro del paquete "rc_sim".

```
roslaunch rc_sim pru.launch
```

El contenido de dicho archivo se muestra en la figura 5.5.6.

Puede apreciarse la sintaxsis sencilla que demarca la creación del nodo "nodo_camara", tipificado por el archivo "nodo_camara" del paquete "rc_sim"; la instrucción 'output="screen"' señala que la salida (por ejemplo notificaciones) sea por pantalla.

Al ejecutar el archivo "pru.launch" utilizando la instrucción "roslaunch" se obtiene la salida por terminal mostrada en la figura 5.5.7.

Figura 5.5.5.: Video capturado por el nodo_camara corriendo sobre el BB

Figura 5.5.6.: Contenido del archivo "pru.launch" del nodo "nodo_cam"

Figura 5.5.7.: Salida en terminal al poner en marcha el nodo ROS "nodo_camara"

Puede apreciarse la conexión exitosa al Master (ubicado en una PC de alias "groovy", pto. 11311) Además, pueden verse en la antedicha figura otros datos relacionados con el funcionamiento del servidor MJPG-Streamer, como puerto, resolución y otros.

5.6. Conclusiones

Como se ha apreciado en este capítulo, los sistemas embebidos poseen ventajas interesantes que pueden ser muy útiles a la hora de diseñar un laboratorio remoto. Entre dichas ventajas figuran: su reducido coste, su modesto tamaño, el consumo notablemente bajo de potencia y la versatilidad que se otorga al programador al permitir, en muchos casos, embeber un sistema operativo completo.

Para el caso que atañe al presente trabajo, se ha mostrado las virtudes de la placa BeagleBone (BB) y se ha mostrado algunas aplicaciones que se han corrido como nodos bajo la arquitectura ROS, con resultados que realmente son buenos.

Hasta lo analizado aquí puede visualizarse una parte del panorama de la presente tesis. Se ha comentado que los laboratorios remotos son un tema de interés en la ingeniería y que por el diseño de los mismo no es descabellado pensarlos como una estructura de sistemas distribuidos y plantearlo dentro de una arquitectura de Middleware Orientado a Mensajes (MOM). Particularmente, se ha hecho la elección de ROS como un sistema plausible a poner en práctica en un caso de estudio de laboratorio remoto y, en el presente capitulo, se ha utilizado una microcomputadora de hardware embebido (BB) en relación con ROS.

En el siguiente capítulo se describirá con detalle el prototipo de laboratorio remoto que se ha implementado, puntualizando los aspectos que resultaron de particular interés en el desarrollo de la presente tesis.

CAPÍTULO VI IMPLEMENTACIÓN DE UN CASO DE ESTUDIO PARA LA ARQUITECTURA PROPUESTA

Con el objeto de probar la eficacia de la arquitectura propuesta (ROS) se construye un escenario de prueba que permita relevar el comportamiento del sistema y evaluar su posibilidad de implementación en laboratorios remotos más complejos orientados a la enseñanza de la ingeniería.

En capítulos anteriores ya se ha establecido un marco teórico que introduce al lector en conceptos importantes relacionados con los laboratorios remotos, las arquitecturas de software para sistemas distribuidos, la plataforma ROS y los sistemas embebidos.

En este capítulo se muestra inicialmente un esquema general del escenario de prueba y la forma de implementar sobre ésta el Middleware Orientado a Mensajes (MOM) escogido: Robot Operating System (ROS). Posteriormente se pesenta una aplicación sencilla de procesamiento de imágenes que ilustra la versatilidad de los nodos ROS al ser utilizados no solo como puntos de procesamiento de datos sino de medición de variables físicas mediante el procesamiento de imágenes. Finalmente se brinda conocimiento de algunas mediciones de desempeño realizadas y se establece la factibilidad de utilizar la arquitectura ROS en la construcción de Laboratorios Remotos.

6.1. Escenario de prueba

El escenario se presenta en la figura 6.1.1. A continuación se dará una rápida explicación general y, posteriormente, sección a sección, se irá ahondando en detalles.

Como puede apreciarse en la Figura 6.1.1, el sistema permite a los alumnos conectarse a través de Internet (puede incluirse validación de usuario) accediendo al servidor principal, el cual posee varias funciones, entre ellas:

1. Gestor de conexiones, usuarios y autenticación (en caso de implementarse)
2. Servidor web
3. Servidor ROSBRIDGE
4. Servidor MJPEG

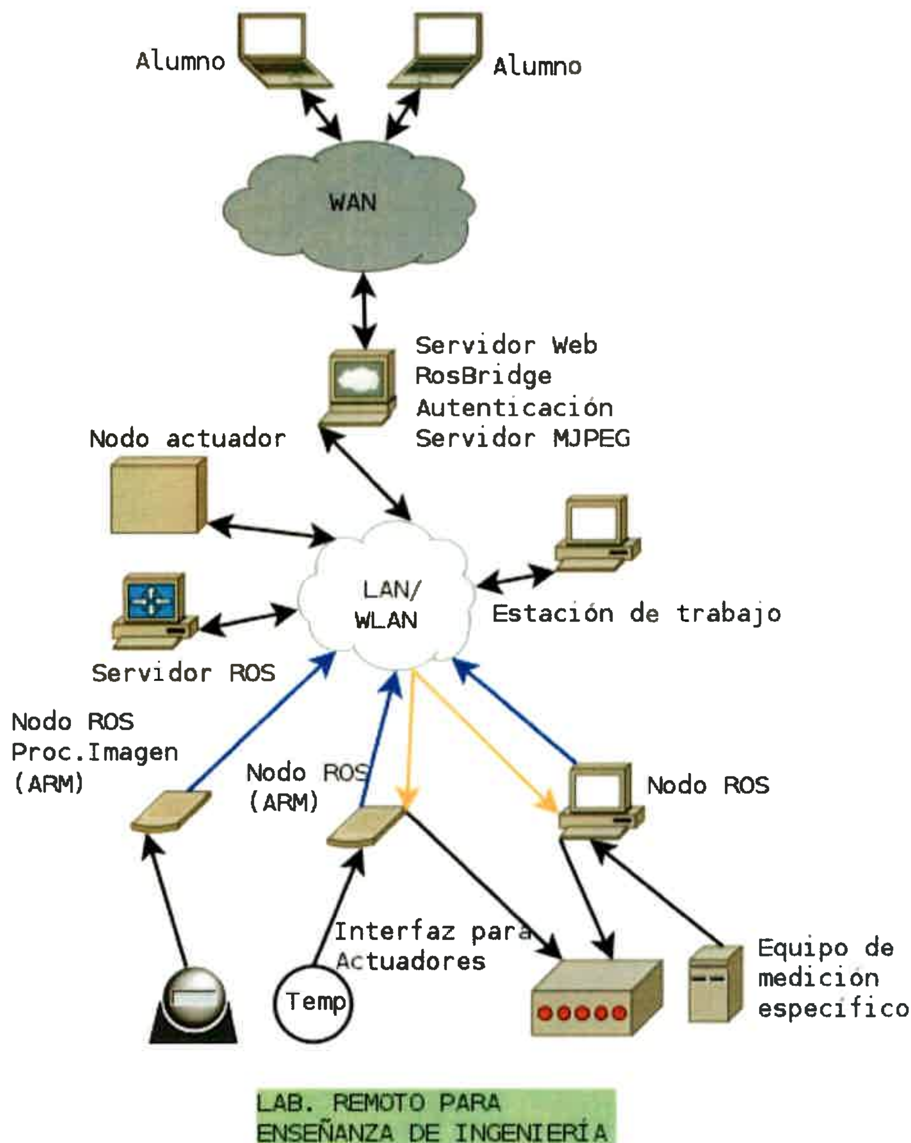


Figura 6.1.1.: Caso de estudio

Resulta especialmente atractivo que la presentación al alumno se hace como una página web, desde donde puede ver, no solo el estado de los sensores y actuadores, sino también controlar a distancia (manual o automáticamente) el experimento. Por tanto, la computadora que se conecte como usuario del sistema sólo necesita disponer de un explorador web actualizado (soporte HTML5 + CSS3 + Javascript).

Puede decirse que el servidor principal sirve de interfaz entre ambos lados de la arquitectura, lado remoto y el laboratorio en planta. Por tanto, una de sus funciones puede ser la gestión de usuarios, de tal manera que sólo accedan los usuarios que están autorizados (por ejemplo los alumnos y docentes de la unidad académica responsable del laboratorio remoto) y a su vez que no accedan dos usuarios en simultáneo, para no generar conflictos. Esta parte del escenario (la autenticación), si bien es importante, puede implementarse posteriormente.

Por otro lado, este servidor también sirve la página web que el lado cliente consulta de forma remota para visualizar y comandar el laboratorio remoto. Dicha web está programada en HTML5 y se utiliza una interfaz que, vía el lenguaje javascript, proporciona la librería ROSBRIDGE. La mencionada librería funciona como puente entre el usuario y la arquitectura ROS subyacente, de tal manera que el primero pueda visualizar y controlar a través de controles web (widgets) los sensores conectados a los nodos ROS. Por otra parte, el servidor MJPEG sirve para enviar un flujo de video comprimido (JPEG cada fotograma) al cliente sobre una vista general de la situación del laboratorio; esto puede resultar orientativo al usuario. Es interesante destacar que este sistema permite al usuario configurar su propia experiencia de laboratorio mediante elegir entre un modo de control automático y manual. En caso de ser automático, se puede configurar un nodo actuador que implemente en local un modelo de control planteado por alumno para manejar el proceso (dentro del esquema PID). Así, vía web el alumno, vé el resultado de su intervención en la respuesta del sistema.

Hasta el momento, y por simplicidad, todas las funciones del "servidor principal" se supusieron en un solo ordenador, pero en realidad podrían ser llevadas a cabo por varias computadoras. De hecho en la implementación que corresponde a las mediciones de este trabajo se implentó de la manera que se muestra en la figura 6.1.2, donde se ve un servidor WEB externo vinculado por una WAN (Internet) al sistema.[1]

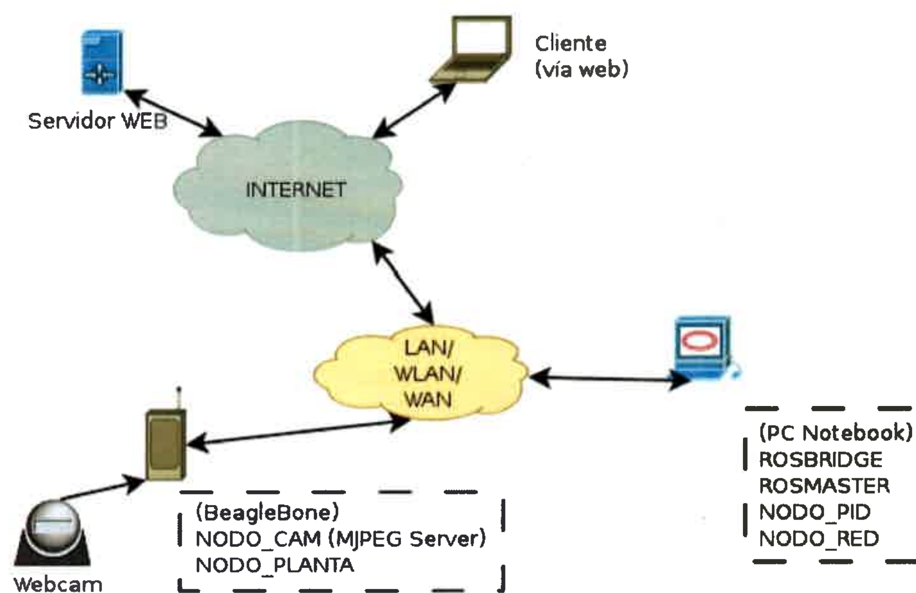


Figura 6.1.2.: "Servidor principal" distribuido en tres dispositivos

En la figura 6.1.1 también puede verse que en el laboratorio hay varias partes independientes conectadas vía red LAN/WLAN, a saber:

[1] Ya se han comenzado las gestiones para trasladar el esquema actual de múltiples servidores conectados a través de internet a servidores internos al campus de la Universidad Nacional de Río Cuarto, con lo cual se puede mejorar la interacción entre las partes por cambiar la red de conexión a una LAN.

- Servidor ROS (Master)
- Nodos Ros
- Estación de Trabajo (cargar / monitorear configuración general)

Por lo que puede apreciarse hasta aquí, es evidente que los nodos ROS son una parte importante y muy activa de la arquitectura, pues están vinculados con sendos sensores y actuadores. En la siguiente sección se hablará de algunos nodos ROS que fueron creados, puestos en marcha y evaluados en el contexto de la presente tesis de Maestría.

Para realizar dicho escenario se creó un paquete ros llamado "rc_sim" el cual tiene el manifiesto (manifest.xml) que se muestra en la figura 6.1.3. Puede apreciarse que las dependencias son otros paquetes incluídos en el metasistema operativo ROS: "std_msgs" (contiene mensajes de tipo estándar, por ejemplo String, Float, Integer, etc.), "rospy" (interacción entre Python y ROS) y "roscpp" (interacción entre C++ y ROS).

```
  GNU nano 2.2.6              File: manifest.xml

<package>
  <description brief="rc_sim">

     rc_sim -> Simulador de llenado de tanque con carga de RC.
               Control PID de RC.
               Comunicaciones via ROS

  </description>
  <author>Ing. Sebastian J. Tosco</author>
  <license>BSD</license>
  <review status="unreviewed" notes=""/>
  <url>http://ros.org/wiki/rc_sim</url>
  <depend package="std_msgs"/>
  <depend package="rospy"/>
  <depend package="roscpp"/>

</package>


^G Get Help  ^O WriteOut  ^R Read Fil  ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Pag  ^U UnCut Tex ^T To Spell
```

Figura 6.1.3.: Manifiesto del paquete "rc_sim"

6.2. Nodos ROS del caso de estudio

Tras consultar con docentes de trayectoria en el campo de las experiencias de laboratorio, se pensó en un caso sumamente simple: controlar el llenado de un tanque de agua. Este

UNIVERSIDAD NACIONAL DE RIO CUARTO
BIBLIOTECA

escenario se tomó como una buena base dado que los docentes asesores veían la posibilidad de ir ampliando el montaje del experimiento mediante otros tanques y válvulas para adecuarlo a consignas complejas en materias afines a la Ingeniería Química.

El sistema de prueba armado tiene 4 nodos:

- Nodo de video (cámara) –> nodo_cam
- Nodo planta (simula RC) –> nodo_planta
- Nodo controlador (PID) –> nodo_pid
- Nodo red (NETEM) –> nodo_red
- Nodo de graficación (opcional - se eliminó en la versión final)

Estos nodos son lanzados desde un archivo central llamado rc_sim.roslaunch, el cual tiene estructura XML; puede apreciarse su contenido en el cuadro 6.2.1. Este archivo es muy importante pues permite con un solo comando lanzar todo el sistema, incluso al estar distribuido en múltiples computadoras.

```
<launch>
        <!--PARAMETROS-->
        <rosparam param="rr">100</rosparam>
        <rosparam param="sp">50</rosparam>
        <rosparam param="Kp">0.075</rosparam>
        <rosparam param="Ki">50</rosparam>
        <rosparam param="Kd">0</rosparam>

        <!--BB-->
        <machine name="bb" address="arm" env-loader="/opt/ros/groovy/env.sh" user="root" password="root" timeout="120"/>


        <node machine="bb" pkg="rc_sim" type="nodo_camara.sh" name="nodo_cam" args="" output="screen">
        </node>

        <node machine="bb" pkg="rc_sim" type="nodo_planta.py" name="nodo_planta" output="screen">
        </node>

        <!--PC-->
        <machine name="pc2" address="192.168.1.250" env-loader="/opt/ros/groovy/env.sh" user="root" password="danilo" />
        <include file="$(find rosbridge_server)/launch/rosbridge_websocket.launch">
           <arg name="port" value="9090"/>
        </include>

        <node machine="pc2" pkg="rc_sim" type="nodo_pid.py" name="nodo_pid" output="screen">
        </node>
        <node machine="pc2" pkg="rc_sim" type="nodo_red.py" name="nodo_red1" output="screen">
        </node>
</launch>

^G Get Help    ^O WriteOut    ^R Read File    ^Y Prev Page    ^K Cut Text     ^C Cur Pos
^X Exit        ^J Justify     ^W Where Is     ^V Next Page    ^U UnCut Text   ^T To Spell
```

Cuadro 6.2.1.: Archivo rc_sim.launch

En cuanto a la estructura interna del archivo "rc_sim.launch" se comenta que allí se aprovecha a establecer los parámetros centrales del sistema, se declaran los nodos a poner en funcionamiento y se identifican los hosts en donde están almacenados dichos nodos. También se actican los nodos servidores, como son el ROS_CORE y el ROS_BRIDGE en el mismo archivo.

6.2.1. Nodo de video

Este nodo corre el servidor MJPG-streamer sobre la placa BeagleBone. Este nodo es lanzado desde una declaración en el archivo de lanzamiento rc_sim.roslaunch que invoca a un archivo de script bash (.sh) que cumple dicho cometido. El contenido de dicho archivo se muestra en la figura 6.2.1.

Figura 6.2.1.: Archivo nodo_camara.sh

Resulta evidente de la figura 6.2.1 que el nodo es sencillamente correr el comando de consola de linux que invoca al MJPG-server; es idéntico al analizado en el ejemplo de la figura 5.5.4 (se repite por comodidad la figura).

Este servidor de video utiliza una compresión MJPEG, la cual es sencilla pero no posee un ratio de compresión destacado.[2] Así, para disminuir el ancho de banda necesario se configura baja calidad de video (160x120 @ 10fps), pero suficiente para la aplicación en cuestión. Igualmente, si se desea bajar más el caudal de tráfico generado puede disminuirse la tasa de refresco (por ejemplo 5fps) La figura 6.2.2 muestra los resultados. Puede observarse una media de 150Kbps, los cuales no son problemáticos en principio para la LAN que interconecta el servidor ROS con el nodo de video y tampoco para una conexion ADSL doméstica típica (Por ej. 3Mbps-downlink)

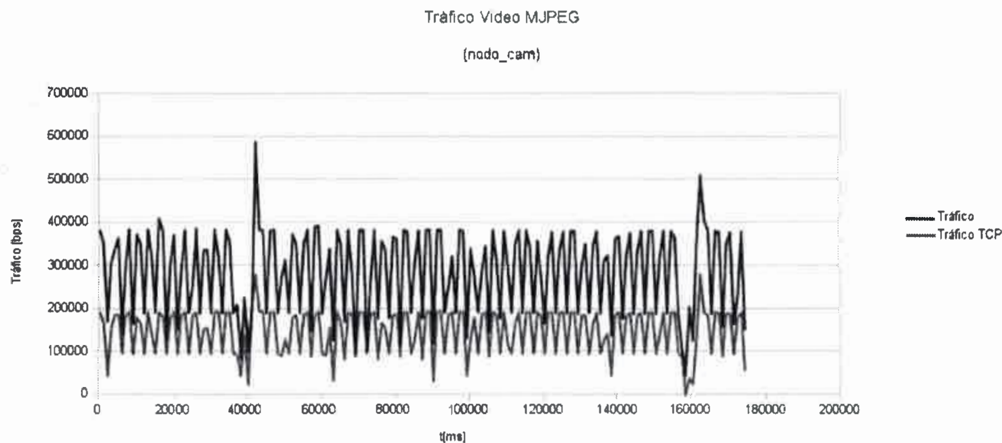


Figura 6.2.2.: Tráfico de video desde el nodo_cam

6.2.2. Nodo planta

Con el objeto de emular la carga de un tanque se utiliza un circuito RC de las siguientes características (figura 6.2.3)

[2]Esquema de compresión intraframe que impone bajos requerimientos de procesamiento y memoria en los dispositivos. Su ratio de compresión y eficiencia es bajo (~1:20) en comparación con MPEG y H.264 (1:50 o mejor) [Fuente: Wikipedia] En la sección Anexo-D se halla un gráfico comparativo interesante sobre este particular.

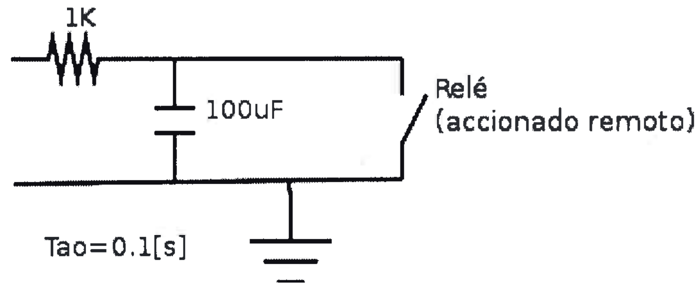


Figura 6.2.3.: Circuito RC

Puede apreciarse que es un sistema lento (Tao=100[ms]). Este sistema RC es apropiado para simular la carga de un tanque y, de hecho, hay antecedentes de este escenario de emulación (Por ejemplo: [41])

Mediante la salida del comando "rostopic hz /Y" puede verse que el sistema está trabajando a una frecuencia promedio de 40hz, con lo cual nos da un muestreo suficiente para el sistema lento (RC) que se posee. Es de interés que la publicación del tópico Y (referido al nivel del tanque) es el elemento rector del sincronismo del sistema, dado que por cada mensaje desde /Y que recibe el PID publica un mensaje de control /U. En la figura 6.2.4 se muestra la situación de cada tópico.

```
danilo@danilo-laptop ~/ros_workspace/rc_sim $ rostopic hz /Y
subscribed to [/Y]
average rate: 40.074
        min: 0.022s max: 0.027s std dev: 0.00087s window: 36
average rate: 39.947
        min: 0.022s max: 0.028s std dev: 0.00084s window: 76
average rate: 40.024
        min: 0.021s max: 0.028s std dev: 0.00091s window: 116
average rate: 39.988
        min: 0.020s max: 0.030s std dev: 0.00113s window: 156
average rate: 40.015
        min: 0.020s max: 0.030s std dev: 0.00111s window: 196
average rate: 40.007
        min: 0.020s max: 0.032s std dev: 0.00124s window: 236
average rate: 40.009
        min: 0.020s max: 0.032s std dev: 0.00121s window: 276
_
```

```
danilo@danilo-laptop ~ $ rostopic hz /U
subscribed to [/U]
average rate: 39.938
        min: 0.020s max: 0.031s std dev: 0.00221s window: 38
average rate: 40.093
        min: 0.019s max: 0.031s std dev: 0.00230s window: 78
average rate: 40.017
        min: 0.018s max: 0.031s std dev: 0.00236s window: 118
average rate: 40.050
        min: 0.018s max: 0.031s std dev: 0.00230s window: 158
average rate: 40.040
        min: 0.018s max: 0.031s std dev: 0.00233s window: 199
average rate: 40.012
        min: 0.018s max: 0.031s std dev: 0.00231s window: 238
average rate: 40.034
        min: 0.018s max: 0.032s std dev: 0.00241s window: 279
```

Figura 6.2.4.: Tasa de publicación de tópicos Nivel(/Y) y Control(/U)

6.2.3. Nodo PID

Como se aprecia en la figura 6.2.4 el nodo PID es el encargado de publicar un mensaje de control que equivale al porcentaje a comandar en el PWM para regular la tensión a bornes del capacitor. La rutina de cálculo del PID está implementada como una clase en Python (Extraída de ejemplo de [42]). En la sección Anexo-E se adjunta código fuente.

Cabe aclarar que los parámetros por defecto del PID se establecen en el archivo de arranque del sistema (rc_sim.launch, ver cuadro 6.1). Estos fueron establecidos en esos valores mediante una estimación intuitiva de respuesta visual, no mediante un método de cálculo de parámetros de la teoría de control, dado que este asunto no compete directamente a los objetivos de esta tesis.

6.2.4. Nodo red

El nodo red ha sido programado mediante un código python elemental y permite la utilización de la librería NETEM modificar retardo y jitter[3] *a la salida del PID - ROS MASTER* (Ver figura 6.1.2). Para más información sobre NETEM ver sección Anexo-F.

6.2.5. Nodo de procesamiento de imágenes

En las primeras pruebas se implementó un nodo de procesamiento de imágenes utilizando la librería OpenCV y sobre la arquietectura ARM. Si bien el nodo era básico, servía para detectar nivel de agua (teñida apropiadamente) en un tanque a partir de la visualización del testigo de nivel con una webcam. Su funcionamiento fue apropiado, aunque se presentaron ciertos problemas a resolver ante situaciones de iluminación no uniforme. Estos inconvenientes no eran imposibles de resolver, pero no se dedicó más tiempo a esta tarea porque no tenía relación directa con los objetivos del presente trabajo. En la figura 6.2.5 se muestra el funcionamiento del nodo de procesamiento para detección de nivel y su visualización en un entorno web (versión anterior, en estado *alfa*).

6.3. Resultados

En las sendas pruebas que se realizaron pudieron establecerse resultados interesantes en cuanto a desempeño y funcionamiento de la arquitectura propuesta para implementar laboratorios remotos. A continuación se describen algunos resultados obtenidos.

6.3.1. Interfaz gráfica html

Las pruebas de uso de la página web demuestran que el diseño HTML5 responsivo es muy apropiado para la aplicación, dado que permite acomodar la estructura de la web a diversos disposivos y tamaños de pantalla. Por ejemplo, la figura 6.3.1 muestra cómo se visualiza la interfaz web cliente en una pantalla de notebook 14" y en una tablet 4.3". En ambos casos es factible para el usuario usar el sistema con facilidad. Además, se dispone de una breve ayuda que permite visualizar el escenario.

[3] Si bien el término "Jitter" puede tener connotaciones ambiguas, por cuestiones de simplicidad en este trabajo se utilizará dicho término para refererirse a la *variación del retardo*.

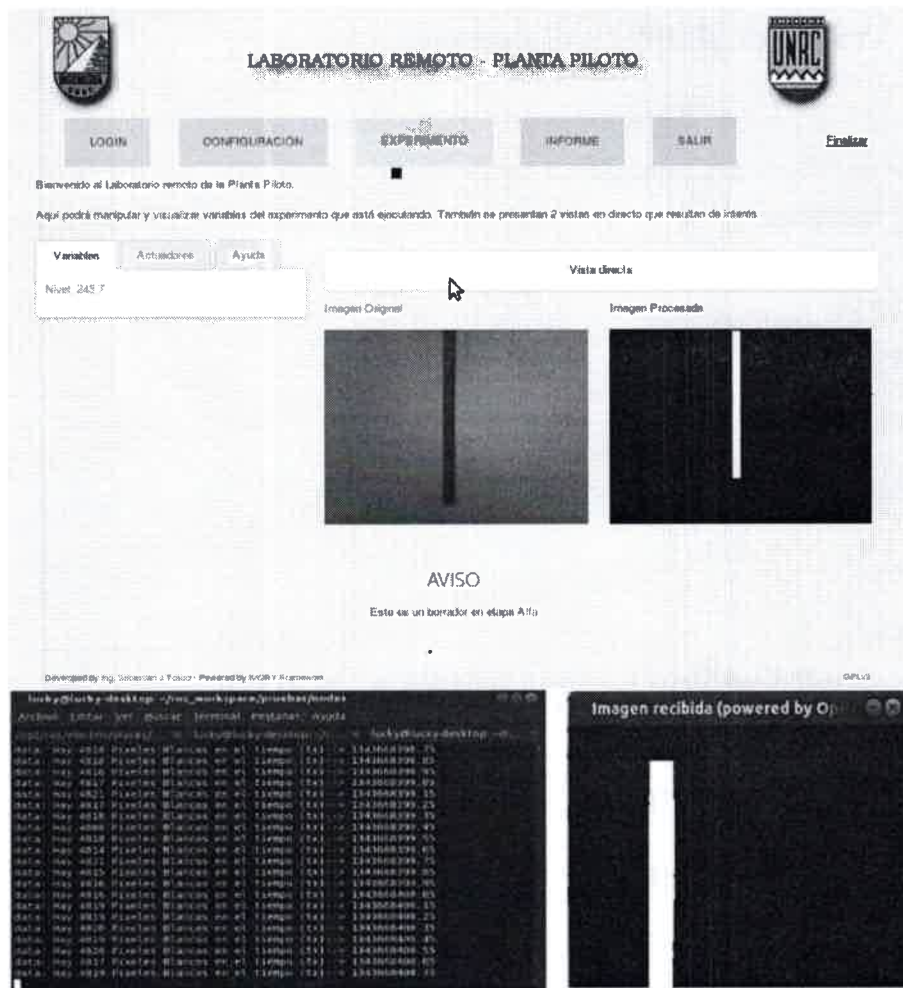


Figura 6.2.5.: Nodo de procesamiento en funcionamiento (entorno web versión *alfa*)

Figura 6.3.1.: Visualización de la interfaz web en 2 dispositivos distintos

6.3.2. Tráfico de datos en el sistema

En la figura 6.3.2 puede notarse una captura de unos 2 minutos de duración del sistema propuesto en funcionamiento. Como puede apreciarse, el tráfico MJPEG de video sigue siendo lo mismo que se obtuvo con anterioridad (unos 150Kbps). El tráfico caracterizado por ROS (pto 9090 correspondiente al ROS_BRIDGE) se estima en unos 180 Kbps y el tráfico total del sistema ronda los 300 Kbps, los cuales no resultan ser privativos en una red moderna.

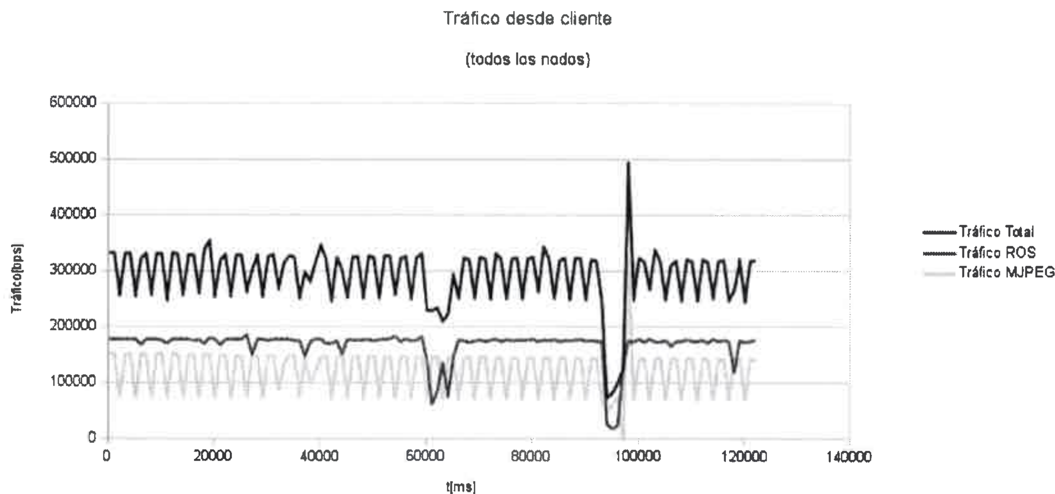


Figura 6.3.2.: Tráfico medido desde el cliente

6.3.3. Influencia de los parámetros de control

Como es de esperarse, la configuración de los parámetros de control utilizados en el sistema tienen gran influencia en el comportamiento del mismo. Así, por ejemplo, se ha podido establecer que el aumento de la tasa de muestreo del sistema aumenta proporcionalmente el tráfico en la red. Por ejemplo, según se mostró en la subsección anterior, el tráfico del sistema es de unos 300 Kbps promedio cuando está funcionando a 40hz. Sin embargo, esta medición sube hasta casi 1Mbps cuando se trabaja a 100hz, lo cual es explicable por el inundado de publicaciones en los tópicos correspondientes. Ahora bien, hay que tener en cuenta que aquí se presenta una decisión de compromiso: muestrear mejor para mayor estabilidad del sistema controlado o ahorrar ancho de banda perdiendo precisión en el control de la respuesta del sistema... Establecer la decisión óptima en este particular se encuentra fuera de los límites propuestos para esta tesis.

En la figura 6.3.3 puede apreciarse cómo el histograma de la respuesta del sistema (variable controlada: nivel del tanque) depende en gran manera de la configuración de los parámetros Kp, Ki, Kd. En dicha figura se muestran 2 casos distintos regulando para el

mismo SetPoint del 50 % del nivel (aprox 0.83). Si bien se probaron varias configuraciones, cabe aclarar que no se buscó realizar el calibrado óptimo del PID dado que eso excede el objetivo de este trabajo, aunque sí se trató de respetar algunas sugerencias sobre la calibración de parámetros de forma manual[4].

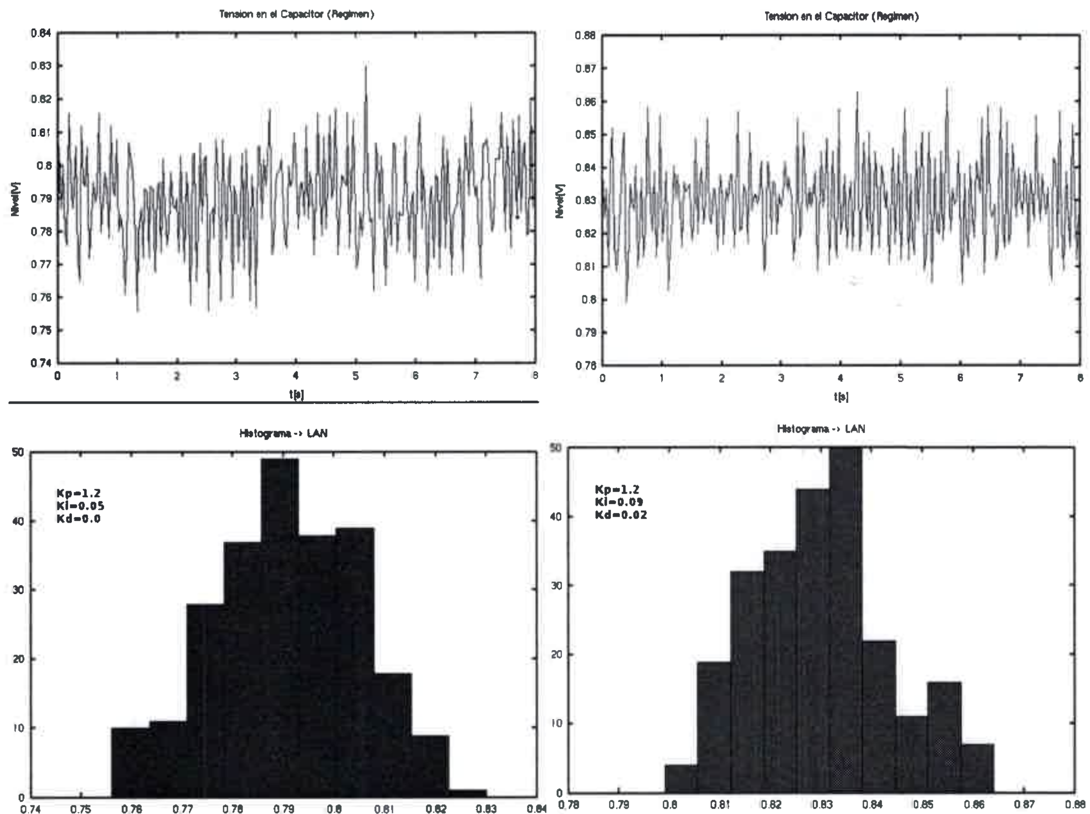


Figura 6.3.3.: Histograma y respuesta del sistema (Nivel) bajo dos configuraciones distintas

6.3.4. Influencia de los retardos

Es conocido el hecho de que las características de retardo en una red LAN y una WLAN(802.11bgn) son distintas. Por ejemplo, una red Lan/WLan tiene valores del estilo que se muestra en el cuadro 6.2, presentando un rtt (round-trip-time) medio ~17ms (estadísticas obtenidas mediante ping con tamaño de paquete de 64B)

Si se prescinde del Wifi, se obtienen las métricas del cuadro 6.3, que de hecho son mucho mejores. En realidad puede verse no sólo la disminución rotunda del retardo, sino también la poca presencia de jitter. Por tanto, para el resto del desarrollo se utilizó red LAN cableada entre la planta y el controlador.

[4] En la siguiente dirección web se pueden encontrar varias sugerencias para una calibración prueba y error: http://www.instrumentacionycontrol.net/cursos-libres/instrumentacion/curso-variadores-de-velocidad/item/197-sintonizacion-pid-en-variadores-de-velocidad.html

— 192.168.1.254 ping statistics —
252 packets transmitted, 246 received, 2 % packet loss, time 251388ms
rtt min/avg/max/mdev = 0.950/16.971/342.539/49.778 ms
Cuadro 6.3.1.: Estadísticas de red WLAN (wifi)

— 192.168.1.254 ping statistics —
131 packets transmitted, 130 received, 0 % packet loss, time 129998ms
rtt min/avg/max/mdev = 0.241/0.361/0.505/0.055 ms
Cuadro 6.3.2.: Estadísticas de la comunicación ROS_MASTER - Planta (sin wifi)

Como puede apreciarse en la figura 6.3.4, la interfaz web del sistema prototipo implementado tiene una sección que permite controlar los parámetros del nodo Red, el cual como dijimos anteriormente utiliza NETEM para emular retardos y jitter (variación del retardo) en la red.

Figura 6.3.4.: Comando de interfaz web para manejo del nodo_red

A continuación se presentan los resultados de simular distintos escenarios de red, esto es distintas convinaciones de retardo y jitter. Para trabajar con prolijidad se buscaron escenarios que tuvieran sentido y que se respaldaran en algún estándar, norma, recomendación o trabajo previo. Así, los cuatro escenarios que se utilizaron son:

1. Red Lan de características similares a las tabuladas en el cuadro 6.3.-
2. Red modelada con lo especificado como Clase 0 por la ITU1541[5] (100ms - 50ms[6]).
3. Red modelada con lo especificado como Clase 1 por la ITU1541 (400ms - 50ms).
4. Red modelada con 200ms-100ms[7].

[5]Se adjunta un cuadro explicativo extraído de dicha recomendación de la ITU en la sección Anexo-G. El documento completo puede conseguirse en la siguiente dirección de internet: http://www.itu.int/ITU-T/recommendations/rec.aspx?rec=11462

[6]A partir de este punto se utilizará la siguiente notación: Delay - Jitter. Así, por ejemplo un retardo de 100[ms] y una variación del mismo de 50[ms] se expresa como 100ms-50ms.

[7]Se sacó el escenario del siguiente trabajo: http://quegrande.org/apuntes/EI/OPT/XR/teoria/08-09/02_-_introduccion_redes_de_comunicaciones.pdf

6.3.4.1. Escenario 1

La figura 6.3.5 contempla la situación 1. Resulta evidente que en estas condiciones se establece un buen control. también se ve la presencia de un pico a la subida y un error de régimen permanente. Ambas cosas pueden ecualizarse con una correcta calibración de los parámetros, cosa que el alumno puede ir probando desde la interfaz web. Un análisis interesante sobre este particular es el tiempo de subida (t_sub) y cómo se puede disminuir mediante una combinación apropiada de las constantes (particularmente agregando "kd"). En la figura 6.3.6 puede ver una disminución del tiempo de subida mediante subir la constante derivativa (kd).

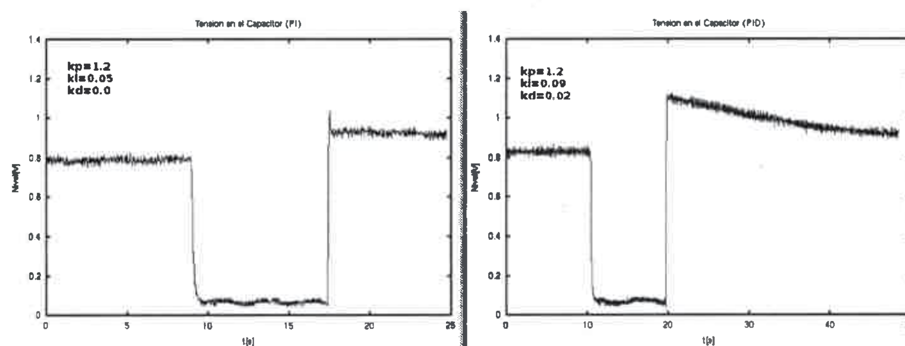


Figura 6.3.5.: Respuesta del sistema (Nivel) a una perturbación para dos configuraciones distintas (escenario 1)

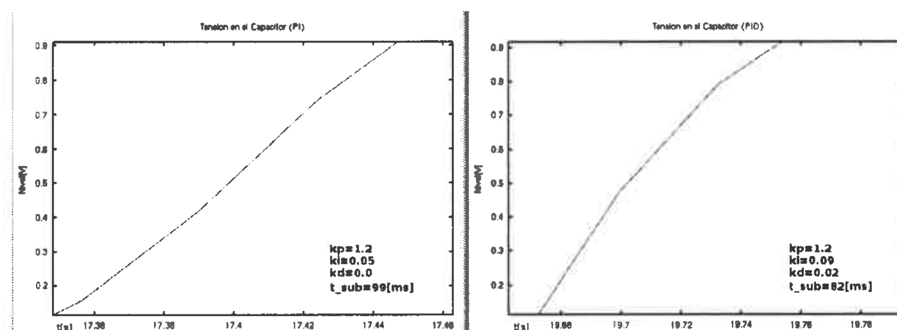


Figura 6.3.6.: Tiempo de subida (escenario 1)

Sin embargo, puede notarse que constituye una propuesta novedosa para la enseñanza de la ingeniería.

A partir de los siguientes escenarios se utiliza la configuración de parámetros correspondiente a la Figura 6.3.6 (a)(color azul):

- kp=1.2
- ki=0.05
- kd=0.0

Se reitera que dicha configuración no pretende ser la óptima, sino más bien un punto de análisis interesante y, en caso de ser operado por un alumno, un punto de partida para realizar un ajuste apropiado del PID.

6.3.4.2. Escenario 2

Puede hacerse otra prueba con valores correspondientes a una Clase 0 de la ITU. La figura 6.3.7 presenta la situación. Resulta evidente que, aunque hay cierto aumento "ruido" (debido a la dificultad para controlar debido al retardo) en comparación con el escenario 1, el sistema puede mantenerse estable (converge al set-point establecido) mediante el control del PID. Viendo el histograma se visualiza bastante "picudo" y centrado en el set-point, excepto por supuesto en los valores correspondientes a la perturbación (valores bajos ~0.1).

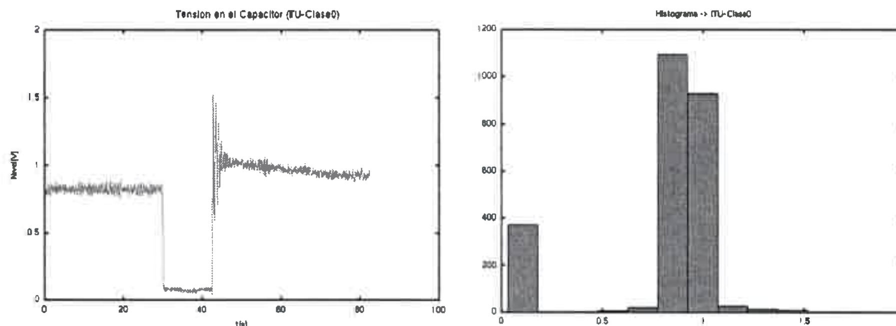


Figura 6.3.7.: Respuesta del sistema (Nivel) a una perturbación e histograma (escenario 2)

6.3.4.3. Escenario 3

Los resultados de otra prueba, ahora con la configuración de red correspondiente a la Clase 1 de la ITU, se muestran plasmados en la figura 6.3.8. Puede apreciarse que el retardo es demasiado grande para poder realizar el control. En una red real, el retardo incluye la pérdida de paquetes que, gracias a TCP que genera sendas retransmisiones, se transforma en retardo. Puede observarse en el histograma que no se destaca ninguna tendencia media, lo cual se explica con la tendencia divergente del sistema en estas condiciones.

6.3.4.4. Escenario 4

Se propuso en las pruebas un escenario intermedio, donde si bien se tiene un retardo intermedio entre las clases 0 y 1 de la ITU (delay≤200ms), se le aumenta al doble de jitter (jitter≤100ms). La figura 6.3.9 muestra los resultados de nivel de tensión e histograma

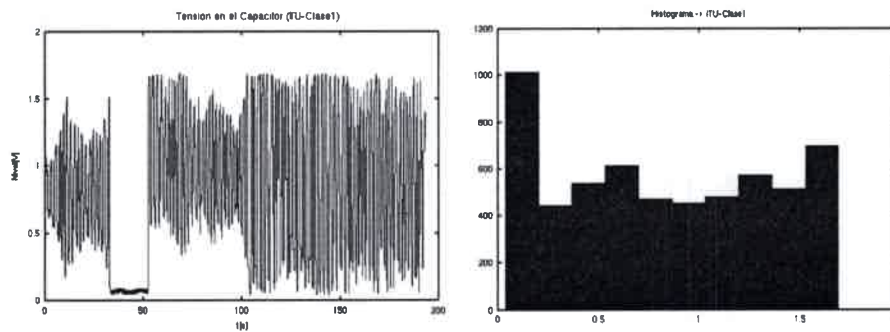


Figura 6.3.8.: Respuesta del sistema (Nivel) a una perturbación e histograma (escenario 3)

correspondiente. Puede verse que si bien la dificultad para realizar el control es mayor que en el escenario 1, no llega a desestabilizarse el sistema. En el histograma se ve un consecuente “ensanchamiento”, lo que corresponde a la mayor fluctuación de los valores de la salida.

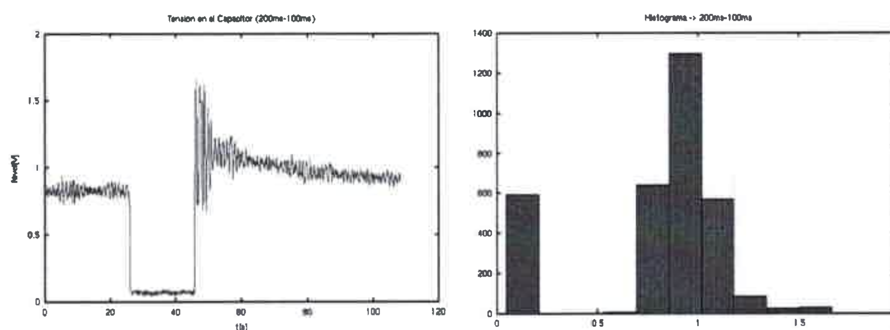


Figura 6.3.9.: Respuesta del sistema (Nivel) a una perturbación e histograma (escenario 4)

6.3.5. Conclusiones

En el presente capítulo se pudo visualizar la potencialidad del MOM ROS y sus capacidades en la implementación de nodos de procesamiento de datos e imágenes. Se presentó un modelo de laboratorio remoto completo que incluía sensado y actuación junto con un nodo de control (PID) y una interfaz gráfica de usuario basada en web.

Con respecto a ésta última, se mostró las ventajas de la programación html 5 con diseño responsivo, lo cual permite que el cliente se independice del tamaño de pantalla del dispositivo que fuese a utilizar y que pueda cómoda y sencillamente acceder al sistema teleoperado.

Por otro lado, se pudo hacer inferencia del comportamiento de la arquitercura basada en ROS diferentes condiciones de red. Así, quedó en evidencia que al armar el laboratorio

será de preferencia armar la red con cable UTP (LAN) y se tratará de evitar las conexiones a nodos mediante WiFi (WLAN), dado que los retardos y jitter inherentes a las arquitecturas inalámbricas.

También, se probó el sistema en cuatro escenario distintos. El sistema demostró responder bien en los escenarios 1, 2 y, con ciertas concesiones, 4. Como contrapartida, el sistema no respondió bien en el escenario 3, dado que los retardos superan la capacidad de control del PID. En esta línea se aclaró también que si, en una red real se perdieran paquetes, esto se traduciría en un aumento del retardo en virtud a las retransmisiones inherentes al comportamiento de TCP, lo que genera un gran Jitter y así el sistema termina perdiendo estabilidad. Esto último afecta en gran manera dado que ROS utiliza un protocolo basado en TCP para realizar las comunicaciones entre los nodos, llamado (aunque parezca redundante) TCPROS. En el Anexo-H se brinda más información sobre el mismo extraída de la wiki oficial.

En cuanto al análisis de ancho de banda requerido, resultó que no se tienen exigencias especiales superiores a una conexión ADSL de mediana-buena calidad (cosa no improbable en estos tiempos; por ejemplo 3M-500Kb ADSL son paquetes asequibles en estos días). Por tanto, queda en evidencia que no es demasiado requerimiento para el alumnado el disponer de un navegador moderno y una conexión a internet decente.

Dicho analisis también reveló que hace falta tener cuidado al escojer la frecuencia de muestreo del sistema (relacionada por supuesto con la constante de tiempo del sistema) y la cantidad de flujos de video a incluir, debido que ambas cosas aumentan los requerimientos de ancho de banda a contratar.

Lo que sí se pudo establecer que en rangos normales de Delay-Jitter ($\leq$ 200ms - 100ms) y con una adecuada selección de tasas de muestreo, para sistemas con constantes de tiempo razonables (muestreables bien con Ts=40HZ o menos), ROS puede ser una plataforma viable para la implementación de un laboratorio remoto semejante al presentado.

CAPÍTULO VII CONCLUSIONES Y TRABAJO A FUTURO

7.1. Introducción

En el presente trabajo se propuso una arquitectura de laboratorio remoto (LR) para enseñanza de las ciencias y la ingeniería. Para lograrlo se hizo un análisis desde lo general hasta lo específico que desembocó en la elección de un middleware de robótica que resulta apropiada para la realización de prácticas experimentales remotas.

Así fue que tras la introducción orientativa del primer capítulo se empezó uno segundo sobre las características de los laboratorios remotos, los lazos de control en los mismos y su vinculación con el concepto de teleoperación. También se dejó establecido que los LR constituyen una propuesta novedosa para la enseñanza de la ingeniería.

En el siguiente capítulo se presentaron las tecnologías implicadas en el diseño de los laboratorios remotos, centrando la atención en los Middleware Orientados a Mensajes (MOM, por sus siglas en inglés). Además, el análisis incluyó una descripción general de las características de los MOM y su funcionamiento genérico, presentando una sucinta comparativa con otra tecnología utilizada en los sistemas distribuídos, a saber RPC. De dicha comparación se destaca que el carácter asincrónico de los MOMs permite aplicaciones donde cada unidad puede operar y responder a peticiones a su ritmo sin necesidad de paralizar el sistema entero. Dicha propiedad reviste especial interés en el desarrollo de la presente temática de tesis.

Para el capítulo cuatro se hizo una elección de un MOM en particular: ROS. Sobre éste pudo apreciarse algunos detalles de su arquitectura interna y su filosofía de funcionamiento. Además, se mostraron en el escrito algunas herramientas dignas de destacar, como el RosBridge y el MJPG-Streamer y se fue explicando cómo fueron utilizadas cada una de ellas en el contexto de la implementación del laboratorio remoto que sirve de caso de estudio en la presente tesis. También, a lo largo del capítulo se dejó claro los aspectos que se tuvieron en cuenta a la hora de decantarse por ROS dentro del abanico de posibilidades de los MOMs. Entre dichas virtudes se recalcan: sus características de rendimiento y funcionamiento, su capacidad de trabajar con lenguaje python y la buena documentación disponible junto a la abundante cantidad de módulos, ejemplos didácticos y paquetes (incluso pilas de software) compartidos.

Habiendo explicado las cuestiones relacionadas con software, en el capítulo cinco se procedió a explicar el papel de los sistemas embebidos como nodos de adquisición en el contexto de los laboratorios remotos. Se mencionó algunas ventajas de los mismos: su reducido coste, su modesto tamaño, el consumo notablemente bajo de potencia y la versatilidad que se otorga al programador al permitir, en muchos casos, embeber un sistema operativo completo. Para el caso que atañe al presente trabajo, se ha mostrado las virtudes de la placa BeagleBone (basada en ARM) y se ha mostrado algunas aplicaciones que se han corrido como nodos bajo la arquitectura ROS, con resultados que realmente son buenos.

Finalmente, el sexto capítulo se puede presentar como la "cereza del postre", dado que muestra la implementación real de un prototipo de laboratorio remoto. En su escrito se pudo visualizar la potencialidad del MOM ROS y sus capacidades en la implementación de nodos de procesamiento de datos e imágenes. Se presentó un modelo de laboratorio remoto completo que incluía sensado y actuación junto con un nodo de control (PID) y una interfaz gráfica de usuario basada en web; dejando sobre esta última constancia de las ventajas de utilizar HTML5-CSS3.

En el capítulo seis, tambien se hizo un análisis de ancho de banda, quedando en evidencia que no hay requerimiento excesivo del mismo, con lo cual el alumnado, disponiendo de un navegador moderno y una conexión a internet decente, no tendría ningún problema para utilizar el LR. Lo que sí se estableció es que en rangos normales de Delay-Jitter ($\leq$ 200ms - 100ms) y con una adecuada selección de tasas de muestreo, para sistemas con constantes de tiempo razonables (muestreables bien con Ts=40HZ o menos), ROS puede ser una *plataforma viable* para la implementación de un laboratorio remoto semejante al presentado.

En el presente capítulo, el cual recoje las conclusiones sobre el trabajo, el autor deja constancia de que, si bien el proceso de formación en la carrera de postgrado requirió un gran esfuerzo, hoy en día se está satisfecho con los resultados obtenidos. Además, el conocer el estado del arte de un tema particular elegido a voluntad e investigado en profundidad fue muy provechoso a la hora de complementar la formación de grado y, seguramente, también permitirá hacer una contribución en la enseñanza de las materias de grado en la que el autor participa.

7.2. Trabajo a futuro

Obviamente, nadie debe creer que lo que ha hecho está acabado y que es inmejorable, dado que eso sería presuntuoso y, según cierto proverbio antiguo, dicha actitud trae deshonra. Así que con sinceridad se procede a comentar algunos aspectos en los que se puede seguir trabajando a futuro.

Por un lado, la idea es poder terminar de implementar el LR de la Planta Piloto del Dpto. de Química de la Fac. de Ingeniería (UNRC), el cual debe terminarse de armar y planificar actividades. Dicha implementación puede ser un primer paso para empezar a ofrecer prácticas de laboratorio a distancia y materias con esta modalidad.

Por el otro, pueden hacerse varios cambios y actualizaciones al sistema. Por ejemplo, podría probarse otro MOM o utilizar la última versión de ROS estable (llamada Hydro). Además, pueden probarse otros sistemas embebidos y comparar desempeños.

Algo muy interesante del sistema ROS es la capacidad de que el usuario cree nodos que vinculen entre sí equipos de muy variadas caracerísticas. A este respecto, algo que resulta interesante como trabajo futuro es mejorar el nodo de procesamiento de imágen y quizás utilizar una vinculación a otro hardware de captación, por ejemplo el Microsoft Kinect. También es posible en esta línea utilizar una mejora en el servidor de video streaming y cambiar la compresión MJPEG por otra de mejores prestaciones, por ejemplo H.264.

En cuanto al ancho de banda, si bien no es grande la exigencia de tráfico presente, puede pensarse en una multiplexación de información en menos tópicos para reducir las comunicaciones, en especial cuando al mismo tiempo se publica información variada de manera sincrónica (por ejemplo siguiendo el muestreo del sistema).

Finalmente, se cree que la utilización de los LR son una herramienta docente interesante que por supuesto, bien instrumentada, puede ser muy beneficiosa en la enseñanza de las ciencias y la ingeniería.

Bibliografía

[1] MJ. Callaghan, J. Harkin, G. Prasad, TM. MeGinnity, LP. Maguire, "Integrated Architecture for Remote Experimentation", IEEE International Conference on System Man and Cybernetics, 2003

[2] M. A. González, J. Adiego, L. F. Sanz, N. Bouab, W. Bouab, J. Mass, "Laboratorios Remotos en la WEB, una Herramienta para la Cooperación al Desarrollo en el Campo de la Educación", Universidad de Valladolid, España

[3] M. Guinaldo, J. Sánchez, S. Dormido, "A Packet-based Network Control System Architecture for Teleoperation and Remote Laboratories", 49th IEEE Conference on Decision and Control, USA, 2010.

[4] S. Hirchie, M. Buss, "Human-Oriented Control for Haptic Teleoperation", Proceeedings of the IEEE, vol 100, N°3, Págs. 623-647.

[5] Y. Liu, C. Chen, M. Meng, "A Study on the Teleoperation of Robot Systems via WWW", Canadian Conference on Electrical and Computer Engineering, IEEE, 2000.

[6] K. Golberg, M.Mascha, "Desktop Teleoperation via the World Wide Web," Proceedings of the 1995 IEEE International Conference on Robotics and Automation, May 1995.

[7] X. Xue, S. X. Yang, M.Q.-H. Meng, "Remote Sensing and Teleoperation of a Mobile Robot via the Internet", Proceedings of the 2005 IEEE International Conference on Information Acquisition, China, 2005.

[8] M.J.H. Lum, J. Rosen, H. King, D.C.W. Friedman, T. S. Lendvay, A.S. Wright, M.N. Sinanan, B. Hannaford, "Teleoperation in Surgical Robotics – Network Latency Effects on Surgical Performance", 31st Annual International Conference of the IEEE EMBS, USA, 2009

[9] B. Jailly, M. Preda, C. Gravier, J. Fayolle, "INTERACTIVE MULTIMEDIA FOR ENGINEERING TELE-OPERATION", Multimedia and Expo (ICME), 2011 IEEE International Conference on, Barcelona, 2011.

[10] E. Litwin, "Tecnologías educativas en tiempos de Internet", Amorrortu editores, Buenos Aires, Argentina, 2005.

[11] C. Culzoni, H. Kofman, P. Lucero, R. Monje. "Propuesta de utilización de internet para laboratorios remotos en la enseñanza de circuitos eléctricos en carreras de ingeniería.", Universidad Tecnológica Nacional; Universidad Nacional del Litoral- Santa Fe

[12] http://es.wikipedia.org/wiki/Web_2.0 (Consultado: 04/2014)

[13] http://yurane.blogspot.com.ar/2011/10/herramientas-de-la-web-20-ventajas-y.html (Consultado: 04/2014)

[14] Curry, Edward, "Message-Oriented Middleware" in "Middleware for Communications." Edited by Mahmoud, Qusay, ISBN 0-470-86206-8, Ed.: John Wiley & Sons Ltd., 2004

[15] McCafferty, Billy, "Message-Based Systems for Maintainable, Asynchronous Development", Blog on devlicio.us.

[16] Wikipedia

[17] Bishop, T., Karne, R. "A survy of Middleware", Computers and Their Applications, 2003 - paginas.fe.up.pt (Consultado: 04/2014)

[18] Bakken, D., "Middleware", http://www.dia.uniroma3.it/~cabibbo/ids/altrui/middleware-bakken.pdf (Consultado: 04/2014)

[19] McCafferty, Billy. Message-Based Systems for Maintainable, Asynchronous Development. On www.sharprobotica.com/page/4/. Año 2010. (Consultado: 04/2014)

[20] Osentoski, S., Jay, G., Crick, C., Jenkins, O.C., "Brown ROS Package: Reproducibility for Shared Experimentation and Learning from Demonstration", American Association for Artificial Intelligence, 2010.

[21] Osentoski, S., Jay, G., Crick, C., Pitzer, B., DuHadway, C., Jenkins, O.C., "Robots as web services: Reproducible experimentation and application development using rosjs", Robotics and Automation (ICRA), 2011 IEEE International Conference on.

[22] Pitzer, B., Osentoski, S., Jay, G., Crick, C., Jenkins, O.C., "PR2 Remote Lab: An environment for remote development and experimentation", Robotics and Automation (ICRA), 2012 IEEE International Conference on

[23] Cruze, B., "Advantages & Disadvantages of Middleware", Computer Software, eHow.com (Consultado: 04/2014)

[24] Elkady, A., Sobh, T. "Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography", Journal of Robotics Volume 2012 (2012), Article ID 959013, 15 pages, 2012.

[25] Hohpe, G., Woolf, B. "Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions", Ed.: Addison-Wesley, 2003.

[26] Tosco, Sebastián J., Corteggiano, F., Broll, M., "Implementación de un esquema de teleoperación utilizando el sistema operativo ros en el contexto de un laboratorio remoto", Mecánica Computacional Vol XXXI, págs. 3741-3749, Salta, Argentina, 13-16 Noviembre 2012

[27] Tosco, Sebastián J., Corteggiano, F., Broll, M., "Teleoperación de un laboratorio remoto para la enseñanza a distancia de la ingeniería", XV Reunión de Trabajo en Procesamiento de la Información y Control, Río Negro, Argentina, 16 al 20 de septiembre de 2013.

[28] http://pr2-remotelab.com/ (Consultado: 04/2014)

[29] Osentoski, S., Pitzer, B., Crick, C., Jay, G., Dong, S., Grollman, D., Suay, H. B., and Jenkins, O. C., "Remote robotics laboratories for learning from demonstration: Enabling user interaction and shared experimentation", International Journal of Social Robotics, 2012.

[30] Fayolle, J., Gravier C., Jailly B., "Collaborative remote laboratory in virtual world", In Proceedings of the 10th WSEAS on Applied Informatics and Communication, Taipei, Taiwan, August 20–22, 2010.

[31] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, A. Ng, "ROS: an open-source Robot Operating System", Stanford University - Willow Garage - University of Southern California.

[32] Sitio oficial de ROS, http://wiki.ros.org (Consultado: 04/2014)

[33] http://barraq.github.io/fOSSa2012/slides.html (Consultado: 04/2014)

[34] https://courses.cs.washington.edu/courses/cse466/12au/calendar/16f-ros_tutorial.pdf (Consultado: 04/2014)

[35] http://www.researchgate.net/post/Differences_between_ROS_and_OROCOS (Consultado: 04/2014)

[36] http://www.quora.com/What-are-the-advantages-and-disadvantages-if-any-of-Robot-Operating-System-ROS (Consultado: 04/2014)

[37] http://sourceforge.net/apps/mediawiki/mjpg-streamer/index.php?title=Main_Page (Consultado: 04/2014)

[38] Página del Simposio Argentino de Sistemas Embebidos, http://www.sase.com.ar/ (Consultado: 04/2014)

[39] Heath, S., "Embedded Systems Design" 2° edition, Ed. Newnes, 2003

[40] http://wiki.ros.org/ROS/CommandLineTools (Consultado: 04/2014)

[41] http://abe-bhaleraolab.age.uiuc.edu/blog/2012/mar/08/pid-control-using-arduino/ (Consultado: 04/2014)

[42] Hugues, J.M., "Real-World Instrumentation with Python", O'Really Media, 2010 (ISBN 978-0-596-80956-0)

Anexos

En esta sección del presente trabajo se adjuntan los anexos. Los mismos se ordenan según se muestra a continuación:

- Anexo A: Conceptos básicos ROS (Catkin)
- Anexo B: Conceptos básicos ROS (Rosbuild)
- Anexo C: Hoja de comandos útiles
- Anexo D: Comparativa entre formatos de video
- Anexo E: Código en Python de la clase *PID* implementada
- Anexo F: Información sobre *netem*
- Anexo G: Definición de Clases –IP QoS (ITU-T)
- Anexo H: Capa de transporte *TCPROS*

Anexo A: Conceptos básicos de Ros (Catkin)

***Getting Started (/ROS/StartGuide)**: Introduction (/ROS/Introduction) | Concepts | Higher-Level Concepts (/ROS/Higher-Level%20Concepts) | Client Libraries (/Client%20Libraries) | Technical Overview (/ROS/Technical%20Overview)*

Contents

1. ROS Filesystem Level
2. ROS Computation Graph Level
3. ROS Community Level
4. Names
    1. Graph Resource Names
    2. Package Resource Names
5. Next

ROS has three levels of concepts: the Filesystem level, the Computation Graph level, and the Community level. These levels and concepts are summarized below and later sections go into each of these in greater detail.

In addition to the three levels of concepts, ROS also defines two types of names (/Names) -- Package Resource Names and Graph Resource Names -- which are discussed below.

Note: since ROS Groovy these wiki pages describe concepts as they relate to the new catkin buildsystem. For older versions of ROS or when using the rosbuild buildsystem, see: rosbuild/ROS/Concepts (/rosbuild /ROS/Concepts)

1. ROS Filesystem Level

The filesystem level concepts mainly cover ROS resources that you encounter on disk, such as:

- **Packages (/Packages)**: Packages are the main unit for organizing software in ROS. A package may contain ROS runtime processes (*nodes*), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together. Packages are the most atomic build item and release item in ROS. Meaning that the most granular thing you can build and release is a package.
- **Metapackages (/Metapackages)**: Metapackages are specialized Packages which only serve to represent a group of related other packages. Most commonly metapackages are used as a backwards compatible place holder for converted rosbuild (/rosbuild) Stacks (/rosbuild /ROS/Concepts#Stacks).
- **Package Manifests (/catkin/package.xml)**: Manifests (`package.xml`) provide metadata about a package, including its name, version, description, license information, dependencies, and other meta information like exported packages. The `package.xml` package manifest is defined in REP-0127 (http://www.ros.org/reps/rep-0127.html).
- **Repositories**: A collection of packages which share a common VCS system. Packages which share a VCS share the same version and can be released together using the catkin release automation tool

bloom (/bloom). Often these repositories will map to converted rosbuild (/rosbuild) Stacks (/rosbuild /ROS/Concepts#Stacks). Repositories can also contain only one package.

- **Message (msg) types (/msg)**: Message descriptions, stored in `my_package/msg/MyMessageType.msg`, define the data structures for messages (/Messages) sent in ROS.
- **Service (srv) types (/srv)**: Service descriptions, stored in `my_package/srv/MyServiceType.srv`, define the request and response data structures for services (/Services) in ROS.

2. ROS Computation Graph Level

The *Computation Graph* is the peer-to-peer network of ROS processes that are processing data together. The basic Computation Graph concepts of ROS are *nodes*, *Master*, *Parameter Server*, *messages*, *services*, *topics*, and *bags*, all of which provide data to the Graph in different ways.

These concepts are implemented in the ros_comm (/ros_comm) repository.

- **Nodes (/Nodes)**: Nodes are processes that perform computation. ROS is designed to be modular at a fine-grained scale; a robot control system usually comprises many nodes. For example, one node controls a laser range-finder, one node controls the wheel motors, one node performs localization, one node performs path planning, one Node provides a graphical view of the system, and so on. A ROS node is written with the use of a ROS client library (/Client%20Libraries), such as roscpp (/roscpp) or rospy (/rospy).
- **Master (/Master)**: The ROS Master provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.
- **Parameter Server (/Parameter%20Server)**: The Parameter Server allows data to be stored by key in a central location. It is currently part of the Master.
- **Messages (/Messages)**: Nodes communicate with each other by passing messages (/Messages). A message is simply a data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structs).
- **Topics (/Topics)**: Messages are routed via a transport system with publish / subscribe semantics. A node sends out a message by *publishing* it to a given topic (/Topics). The topic is a name (/Names) that is used to identify the content of the message. A node that is interested in a certain kind of data will *subscribe* to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence. The idea is to decouple the production of information from its consumption. Logically, one can think of a topic as a strongly typed message bus. Each bus has a name, and anyone can connect to the bus to send or receive messages as long as they are the right type.
- **Services (/Services)**: The publish / subscribe model is a very flexible communication paradigm, but its many-to-many, one-way transport is not appropriate for request / reply interactions, which are often required in a distributed system. Request / reply is done via services (/Services), which are

defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name (/Names) and a client uses the service by sending the request message and awaiting the reply. ROS client libraries generally present this interaction to the programmer as if it were a remote procedure call.

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

The ROS Master (/Master) acts as a nameservice in the ROS Computation Graph. It stores topics (/Topics) and services (/Services) registration information for ROS nodes (/Nodes). Nodes communicate with the Master to report their registration information. As these nodes communicate with the Master, they can receive information about other registered nodes and make connections as appropriate. The Master will also make callbacks to these nodes when this registration information changes, which allows nodes to dynamically create connections as new nodes are run.

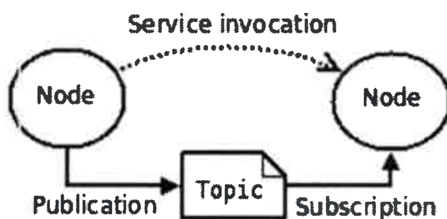
Nodes connect to other nodes directly; the Master only provides lookup information, much like a DNS server. Nodes that subscribe to a topic will request connections from nodes that publish that topic, and will establish that connection over an agreed upon connection protocol. The most common protocol used in a ROS is called TCPROS (/ROS/TCPROS), which uses standard TCP/IP sockets.

This architecture allows for decoupled operation, where the names (/Names) are the primary means by which larger and more complex systems can be built. Names have a very important role in ROS: nodes, topics, services, and parameters all have names. Every ROS client library (/Client%20Libraries) supports command-line remapping of names (/Remapping%20Arguments), which means a compiled program can be reconfigured at runtime to operate in a different Computation Graph topology.

For example, to control a Hokuyo laser range-finder, we can start the hokuyo_node (/hokuyo_node) driver, which talks to the laser and publishes sensor_msgs/LaserScan (http://docs.ros.org/api/sensor_msgs /html/msg/LaserScan.html) messages on the `scan` topic. To process that data, we might write a node using laser_filters (/laser_filters) that subscribes to messages on the `scan` topic. After subscription, our filter would automatically start receiving messages from the laser.

Note how the two sides are decoupled. All the `hokuyo_node` node does is publish scans, without knowledge of whether anyone is subscribed. All the filter does is subscribe to scans, without knowledge of whether anyone is publishing them. The two nodes can be started, killed, and restarted, in any order, without inducing any error conditions.

Later we might add another laser to our robot, so we need to reconfigure our system. All we need to do is *remap* the names that are used. When we start our first `hokuyo_node`, we could tell it instead to remap `scan` to `base_scan`, and do the same with our filter node. Now, both of these nodes will communicate using the `base_scan` topic instead and not hear messages on the `scan` topic. Then we can just start another `hokuyo_node` for the new laser range finder.



ROS_basic_concepts.dia (/ROS/Concepts?action=AttachFile&do=view&target=ROS_basic_concepts.dia)

3. ROS Community Level

The ROS Community Level concepts are ROS resources that enable separate communities to exchange software and knowledge. These resources include:

- **Distributions (/Distributions)**: ROS Distributions are collections of versioned stacks (/Stacks) that you can install. Distributions play a similar role to Linux distributions: they make it easier to install a collection of software, and they also maintain consistent versions across a set of software.
- **Repositories (/Repositories)**: ROS relies on a federated network of code repositories, where different institutions can develop and release their own robot software components.
- **The ROS Wiki (/Documentation)**: The ROS community Wiki is the main forum for documenting information about ROS. Anyone can sign up for an account and contribute their own documentation, provide corrections or updates, write tutorials, and more.
- **Bug Ticket System**: Please see Tickets (/Tickets) for information about file tickets.
- **Mailing Lists (/Mailing%20Lists)**: The ros-users mailing list (/Mailing%20Lists) is the primary communication channel about new updates to ROS, as well as a forum to ask questions about ROS software.
- **ROS Answers (http://answers.ros.org)**: A Q&A site for answering your ROS-related questions.
- **Blog (http://www.willowgarage.com/blog)**: The Willow Garage Blog (http://www.willowgarage.com/blog) provides regular updates, including photos and videos.

Contents

1. Names
   1. Graph Resource Names
      1. Valid Names
      2. Resolving
      3. Remapping
   2. Package Resource Names
      1. Valid Names

4. Names

4.1 Graph Resource Names

Graph Resource Names provide a hierarchical naming structure that is used for all resources in a ROS

Computation Graph, such as Nodes (/Nodes), Parameters (/Parameter%20Server), Topics (/Topics), and Services (/Services). These names are very powerful in ROS and central to how larger and more complicated systems are composed in ROS, so it is critical to understand how these names work and how you can manipulate them.

Before we describe names further, here are some example names:

- / (the global namespace)
- `/foo`
- `/stanford/robot/name`
- `/wg/node1`

Graph Resource Names are an important mechanism in ROS for providing encapsulation. Each resource is defined within a namespace, which it may share with many other resources. In general, resources can *create* resources within their namespace and they can *access* resources within or above their own namespace. Connections can be made between resources in distinct namespaces, but this is generally done by integration code above both namespaces. This encapsulation isolates different portions of the system from accidentally grabbing the wrong named resource or globally hijacking names.

Names are resolved relatively, so resources do not need to be aware of which namespace they are in. This simplifies programming as nodes that work together can be written as if they are all in the top-level namespace. When these Nodes are integrated into a larger system, they can be *pushed down* into a namespace that defines their collection of code. For example, one could take a Stanford demo and a Willow Garage demo and merge them into a new demo with `stanford` and wg subgraphs. If both demos had a Node named 'camera', they would not conflict. Tools (e.g. graph visualization) as well as parameters (e.g. `demo_name`) that need to be visible to the entire graph can be created by top-level Nodes.

4.1.1 Valid Names

A valid name has the following characteristics:

1. First character is an alpha character ([a-z|A-Z]), tilde (~) or forward slash (/)
2. Subsequent characters can be alphanumeric ([0-9|a-z|A-Z]), underscores (_), or forward slashes (/)

Exception: base names (described below) cannot have forward slashes (/) or tildes (~) in them.

4.1.2 Resolving

There are four types of Graph Resource Names in ROS: *base*, *relative*, *global*, and *private*, which have the following syntax:

- `base`
- `relative/name`
- `/global/name`
- `~private/name`

By default, resolution is done *relative* to the node's namespace. For example, the node `/wg/node1` has the namespace `/wg`, so the name `node2` will resolve to `/wg/node2`.

Names with no namespace qualifiers whatsoever are *base* names. Base names are actually a subclass of relative names and have the same resolution rules. Base names are most frequently used to initialize the node name.

Names that start with a "/" are *global* -- they are considered fully resolved. Global names should be avoided as much as possible as they limit code portability.

Names that start with a "~" are *private*. They convert the node's name into a namespace. For example, `node1` in namespace `/wg/` has the private namespace `/wg/node1`. Private names are useful for passing parameters to a specific node via the parameter server.

Here are some name resolution examples:

Node	**Relative** (default)	**Global**	**Private**
`/node1`	`bar` -> `/bar`	`/bar` -> `/bar`	`~bar` -> `/node1/bar`
`/wg/node2`	`bar` -> `/wg/bar`	`/bar` -> `/bar`	`~bar` -> `/wg/node2/bar`
`/wg/node3`	`foo/bar` -> `/wg/foo/bar`	`/foo/bar` -> `/foo/bar`	`~foo/bar` -> `/wg/node3/foo/bar`

4.1.3 Remapping

Any name within a ROS Node can be remapped when the Node is launched at the command-line. For more information on this feature, see Remapping Arguments (/Remapping%20Arguments).

4.2 Package Resource Names

Package Resource Names are used in ROS with Filesystem-Level concepts to simplify the process of referring to files and data types on disk. Package Resource Names are very simple: they are just the name of the Package (/Packages) that the resource is in plus the name of the resource. For example, the name "std_msgs/String" refers to the "String" message type in the "std_msgs" Package.

Some of the ROS-related files that may be referred to using Package Resource Names include:

- Message (msg) types (/msg)
- Service (srv) types (/srv)
- Node types (/Nodes)

Package Resource Names are very similar to file paths, except they are much shorter. This is due to the ability of ROS to locate Packages on disk and make additional assumptions about their contents. For example, Message descriptions are always stored in the `msg` subdirectory and have the `.msg` extension, so `std_msgs/String` is shorthand for `path/to/std_msgs/msg/String.msg`. Similarly, the Node type `foo/bar` is equivalent to searching for a file named `bar` in Package `foo` with executable permissions.

4.2.1 Valid Names

Package Resource Names have strict naming rules as they are often used in auto-generated code. For this reason, a ROS package (/Packages) cannot have special characters other than an underscore, and they must start with an alphabetical character. A valid name has the following characteristics:

1. First character is an alpha character ([a-z|A-Z])
2. Subsequent characters can be alphanumeric ([0-9|a-z|A-Z]), underscores (_) or a forward slash (/)
3. There is at most one forward slash ('/').

5. Next

Higher-Level Concepts (/ROS/Higher-Level%20Concepts)

Except where otherwise noted, the ROS wiki is licensed under the
Creative Commons Attribution 3.0 (http://creativecommons.org/licenses/by/3.0/) | Find us on Google+ (https://plus.google.com/113789706402978299308)

Wiki: ROS/Concepts (last edited 2013-12-23 14:56:54 by TullyFoote (/TullyFoote))

(http://www.osrfoundation.org)

Anexo B: Conceptos básicos de Ros (Rosbuild)

__Getting Started (/rosbuild/ROS/StartGuide)__: Introduction (/rosbuild/ROS/Introduction) | Concepts | Higher-Level Concepts (/rosbuild/ROS/Higher-Level%20Concepts) | Client Libraries (/rosbuild/Client%20Libraries) | Technical Overview (/rosbuild/ROS/Technical%20Overview)

Note: This is a legacy, rosbuild based, version of the ROS Overview

Contents

1. ROS Filesystem Level
2. ROS Computation Graph Level
3. ROS Community Level
4. Names
    1. Graph Resource Names
    2. Package Resource Names
5. Next

ROS has three levels of concepts: the Filesystem level, the Computation Graph level, and the Community level. These levels and concepts are summarized below and later sections go into each of these in greater detail.

In addition to the three levels of concepts, ROS also defines two types of names (/Names) -- Package Resource Names and Graph Resource Names -- which are discussed below.

1. ROS Filesystem Level

The filesystem level concepts are ROS resources that you encounter on disk, such as:

- **Packages (/rosbuild/Packages)**: Packages are the main unit for organizing software in ROS. A package may contain ROS runtime processes (*nodes*), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together.
- **Manifests (/rosbuild/Manifest)**: Manifests (`manifest.xml`) provide metadata about a package, including its license information and dependencies, as well as language-specific information such as compiler flags.
- **Stacks (/rosbuild/Stacks)**: Stacks are collections of packages that provide aggregate functionality, such as a "navigation stack." Stacks are also how ROS software is released and have associated version numbers
- **Stack Manifests (/rosbuild/Stack%20Manifest)**: Stack manifests (`stack.xml`) provide data about a stack, including its license information and its dependencies on other stacks.
- **Message (msg) types (/rosbuild/msg)**: Message descriptions, stored in `my_package/msg/MyMessageType.msg`, define the data structures for messages (/Messages) sent in ROS.
- **Service (srv) types (/rosbuild/srv)**: Service descriptions, stored in `my_package/srv/MyServiceType.srv`, define the request and response data structures for services (/Services) in ROS.

2. ROS Computation Graph Level

The *Computation Graph* is the peer-to-peer network of ROS processes that are processing data together. The basic Computation Graph concepts of ROS are *nodes*, *Master*, *Parameter Server*, *messages*, *services*, *topics*, and *bags*, all of which provide data to the Graph in different ways.

These concepts are implemented in the ros_comm (/ros_comm) stack.

- **Nodes (/Nodes)**: Nodes are processes that perform computation. ROS is designed to be modular at a fine-grained scale; a robot control system usually comprises many nodes. For example, one node controls a laser range-finder, one node controls the wheel motors, one node performs localization, one node performs path planning, one Node provides a graphical view of the system, and so on. A ROS node is written with the use of a ROS client library (/rosbuild/Client%20Libraries), such as roscpp (/roscpp) or rospy (/rospy).
- **Master (/Master)**: The ROS Master provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.
- **Parameter Server (/Parameter%20Server)**: The Parameter Server allows data to be stored by key in a central location. It is currently part of the Master.
- **Messages (/Messages)**: Nodes communicate with each other by passing messages (/Messages). A message is simply a data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structs).
- **Topics (/Topics)**: Messages are routed via a transport system with publish / subscribe semantics. A node sends out a message by *publishing* it to a given topic (/Topics). The topic is a name (/Names) that is used to identify the content of the message. A node that is interested in a certain kind of data will *subscribe* to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence. The idea is to decouple the production of information from its consumption. Logically, one can think of a topic as a strongly typed message bus. Each bus has a name, and anyone can connect to the bus to send or receive messages as long as they are the right type.
- **Services (/Services)**: The publish / subscribe model is a very flexible communication paradigm, but its many-to-many, one-way transport is not appropriate for request / reply interactions, which are often required in a distributed system. Request / reply is done via services (/Services), which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name (/Names) and a client uses the service by sending the request message and awaiting the reply. ROS client libraries generally present this interaction to the programmer as if it were a remote procedure call.
- **Bags (/Bags)**: Bags are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms.

The ROS Master (/Master) acts as a nameservice in the ROS Computation Graph. It stores topics (/Topics)

and services (/Services) registration information for ROS nodes (/Nodes). Nodes communicate with the Master to report their registration information. As these nodes communicate with the Master, they can receive information about other registered nodes and make connections as appropriate. The Master will also make callbacks to these nodes when this registration information changes, which allows nodes to dynamically create connections as new nodes are run.

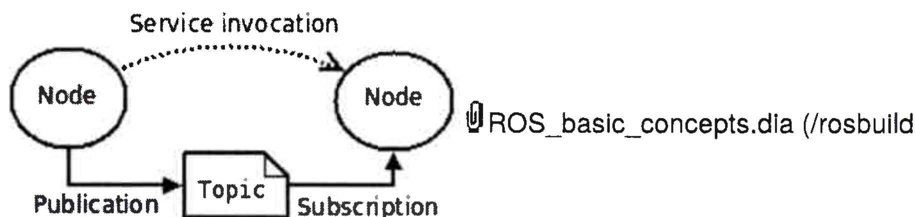
Nodes connect to other nodes directly; the Master only provides lookup information, much like a DNS server. Nodes that subscribe to a topic will request connections from nodes that publish that topic, and will establish that connection over an agreed upon connection protocol. The most common protocol used in a ROS is called TCPROS (/ROS/TCPROS), which uses standard TCP/IP sockets.

This architecture allows for decoupled operation, where the names (/Names) are the primary means by which larger and more complex systems can be built. Names have a very important role in ROS: nodes, topics, services, and parameters all have names. Every ROS client library (/rosbuild/Client%20Libraries) supports command-line remapping of names (/Remapping%20Arguments), which means a compiled program can be reconfigured at runtime to operate in a different Computation Graph topology.

For example, to control a Hokuyo laser range-finder, we can start the hokuyo_node (/hokuyo_node) driver, which talks to the laser and publishes sensor_msgs/LaserScan (http://docs.ros.org/api/sensor_msgs/html/msg/LaserScan.html) messages on the `scan` topic. To process that data, we might write a node using laser_filters (/laser_filters) that subscribes to messages on the `scan` topic. After subscription, our filter would automatically start receiving messages from the laser.

Note how the two sides are decoupled. All the `hokuyo_node` node does is publish scans, without knowledge of whether anyone is subscribed. All the filter does is subscribe to scans, without knowledge of whether anyone is publishing them. The two nodes can be started, killed, and restarted, in any order, without inducing any error conditions.

Later we might add another laser to our robot, so we need to reconfigure our system. All we need to do is *remap* the names that are used. When we start our first `hokuyo_node`, we could tell it instead to remap `scan` to `base_scan`, and do the same with our filter node. Now, both of these nodes will communicate using the `base_scan` topic instead and not hear messages on the `scan` topic. Then we can just start another `hokuyo_node` for the new laser range finder.



ROS_basic_concepts.dia (/rosbuild/ROS/Concepts?action=AttachFile&do=upload_form&ticket=00538e628d.517dfe243855214881ef35aea464792e0c15d46b&target=ROS_basic_concepts.dia)

3. ROS Community Level

The ROS Community Level concepts are ROS resources that enable separate communities to exchange software and knowledge. These resources include:

- **Distributions (/Distributions)**: ROS Distributions are collections of versioned stacks (/rosbuild /Stacks) that you can install. Distributions play a similar role to Linux distributions: they make it easier to install a collection of software, and they also maintain consistent versions across a set of software.
- **Repositories (/Repositories)**: ROS relies on a federated network of code repositories, where different institutions can develop and release their own robot software components.
- **The ROS Wiki (/Documentation)**: The ROS community Wiki is the main forum for documenting information about ROS. Anyone can sign up for an account and contribute their own documentation, provide corrections or updates, write tutorials, and more.
- **Bug Ticket System**: Please see Tickets (/Tickets) for information about file tickets.
- **Mailing Lists (/Mailing%20Lists)**: The ros-users mailing list (/Mailing%20Lists) is the primary communication channel about new updates to ROS, as well as a forum to ask questions about ROS software.
- **ROS Answers (http://answers.ros.org)**: A Q&A site for answering your ROS-related questions.
- **Blog (http://www.willowgarage.com/blog)**: The Willow Garage Blog (http://www.willowgarage.com/blog) provides regular updates, including photos and videos.

Contents

1. Names
   1. Graph Resource Names
      1. Valid Names
      2. Resolving
      3. Remapping
   2. Package Resource Names
      1. Valid Names

4. Names

4.1 Graph Resource Names

Graph Resource Names provide a hierarchical naming structure that is used for all resources in a ROS Computation Graph, such as Nodes (/Nodes), Parameters (/Parameter%20Server), Topics (/Topics), and Services (/Services). These names are very powerful in ROS and central to how larger and more complicated systems are composed in ROS, so it is critical to understand how these names work and how you can manipulate them.

Before we describe names further, here are some example names:

- / (the global namespace)
- `/foo`
- `/stanford/robot/name`
- `/wg/node1`

Graph Resource Names are an important mechanism in ROS for providing encapsulation. Each resource is

defined within a namespace, which it may share with many other resources. In general, resources can *create* resources within their namespace and they can *access* resources within or above their own namespace. Connections can be made between resources in distinct namespaces, but this is generally done by integration code above both namespaces. This encapsulation isolates different portions of the system from accidentally grabbing the wrong named resource or globally hijacking names.

Names are resolved relatively, so resources do not need to be aware of which namespace they are in. This simplifies programming as nodes that work together can be written as if they are all in the top-level namespace. When these Nodes are integrated into a larger system, they can be *pushed down* into a namespace that defines their collection of code. For example, one could take a Stanford demo and a Willow Garage demo and merge them into a new demo with `stanford` and wg subgraphs. If both demos had a Node named 'camera', they would not conflict. Tools (e.g. graph visualization) as well as parameters (e.g. `demo_name`) that need to be visible to the entire graph can be created by top-level Nodes.

4.1.1 Valid Names

A valid name has the following characteristics:

1. First character is an alpha character ([a-z|A-Z]), tilde (~) or forward slash (/)
2. Subsequent characters can be alphanumeric ([0-9|a-z|A-Z]), underscores (_), or forward slashes (/)

Exception: base names (described below) cannot have forward slashes (/) or tildes (~) in them.

4.1.2 Resolving

There are four types of Graph Resource Names in ROS: *base*, *relative*, *global*, and *private*, which have the following syntax:

- `base`
- `relative/name`
- `/global/name`
- `~private/name`

By default, resolution is done *relative* to the node's namespace. For example, the node `/wg/node1` has the namespace `/wg`, so the name `node2` will resolve to `/wg/node2`.

Names with no namespace qualifiers whatsoever are *base* names. Base names are actually a subclass of relative names and have the same resolution rules. Base names are most frequently used to initialize the node name.

Names that start with a "/" are *global* -- they are considered fully resolved. Global names should be avoided as much as possible as they limit code portability.

Names that start with a "~" are *private*. They convert the node's name into a namespace. For example, `node1` in namespace `/wg/` has the private namespace `/wg/node1`. Private names are useful for passing parameters to a specific node via the parameter server.

Here are some name resolution examples:

Node	Relative (default)	Global	Private
/node1	bar -> /bar	/bar -> /bar	~bar -> /node1/bar
/wg/node2	bar -> /wg/bar	/bar -> /bar	~bar -> /wg/node2/bar
/wg/node3	foo/bar -> /wg/foo/bar	/foo/bar -> /foo/bar	~foo/bar -> /wg/node3 /foo/bar

4.1.3 Remapping

Any name within a ROS Node can be remapped when the Node is launched at the command-line. For more information on this feature, see Remapping Arguments (/Remapping%20Arguments).

4.2 Package Resource Names

Package Resource Names are used in ROS with Filesystem-Level concepts to simplify the process of referring to files and data types on disk. Package Resource Names are very simple: they are just the name of the Package (/Packages) that the resource is in plus the name of the resource. For example, the name "std_msgs/String" refers to the "String" message type in the "std_msgs" Package.

Some of the ROS-related files that may be referred to using Package Resource Names include:

- Message (msg) types (/msg)
- Service (srv) types (/srv)
- Node types (/Nodes)

Package Resource Names are very similar to file paths, except they are much shorter. This is due to the ability of ROS to locate Packages on disk and make additional assumptions about their contents. For example, Message descriptions are always stored in the `msg` subdirectory and have the `.msg` extension, so `std_msgs/String` is shorthand for `path/to/std_msgs/msg/String.msg`. Similarly, the Node type `foo/bar` is equivalent to searching for a file named `bar` in Package `foo` with executable permissions.

4.2.1 Valid Names

Package Resource Names have strict naming rules as they are often used in auto-generated code. For this reason, a ROS package (/Packages) cannot have special characters other than an underscore, and they must start with an alphabetical character. A valid name has the following characteristics:

1. First character is an alpha character ([a-z|A-Z])
2. Subsequent characters can be alphanumeric ([0-9|a-z|A-Z]), underscores (_) or a forward slash (/)
3. There is at most one forward slash ('/').

5. Next

Higher-Level Concepts (/rosbuild/ROS/Higher-Level%20Concepts)

Except where otherwise noted, the ROS wiki is licensed under the

Wiki: rosbuild/ROS/Concepts (last edited 2013-12-23 15:02:58 by TullyFoote (/TullyFoote))

Creative Commons Attribution 3.0 (http://creativecommons.org/licenses/by/3.0/) | Find us on Google+ (https://plus.google.com/113789706402978299308)

(http://www.osrfoundation.org)

Anexo C: Hoja de comandos útiles

ROS Cheat Sheet

Filesystem Command-line Tools

`rospack/rosstack`	A tool inspecting packages/stacks.
`roscd`	Changes directories to a package or stack.
`rosls`	Lists package or stack information.
`roscreate-pkg`	Creates a new ROS package.
`roscreate-stack`	Creates a new ROS stack.
`rosdep`	Installs ROS package system dependencies.
`rosmake`	Builds a ROS package.
`roswtf`	Displays a errors and warnings about a running ROS system or launch file.
`rxdeps`	Displays package structure and dependencies.

Usage:
```
$ rospack find [package]
$ roscd [package[/subdir]]
$ rosls [package[/subdir]]
$ roscreate-pkg [package_name]
$ rosmake [package]
$ rosdep install [package]
$ roswtf or roswtf [file]
$ rxdeps [options]
```

Common Command-line Tools

roscore

A collection of nodes and programs that are pre-requisites of a ROS-based system. You must have a roscore running in order for ROS nodes to communicate.

roscore is currently defined as:
```
master
parameter server
rosout
```

Usage:
```
$ roscore
```

rosmsg/rossrv

rosmsg/rossrv displays Message/Service (msg/srv) data structure definitions.

Commands:

`rosmsg show`	Display the fields in the msg.
`rosmsg users`	Search for code using the msg.
`rosmsg md5`	Display the msg md5 sum.
`rosmsg package`	List all the messages in a package.
`rosnode packages`	List all the packages with messages.

Examples:

Display the Pose msg:
```
$ rosmsg show Pose
```
List the messages in nav_msgs:
```
$ rosmsg package nav_msgs
```
List the files using sensor_msgs/CameraInfo:
```
$ rosmsg users sensor_msgs/CameraInfo
```

rosrun

rosrun allows you to run an executable in an arbitrary package without having to cd (or roscd) there first.

Usage:
```
$ rosrun package executable
```

Example:

Run turtlesim:
```
$ rosrun turtlesim turtlesim_node
```

rosnode

Displays debugging information about ROS nodes, including publications, subscriptions and connections.

Commands:

`rosnode ping`	Test connectivity to node.
`rosnode list`	List active nodes.
`rosnode info`	Print information about a node.
`rosnode machine`	List nodes running on a particular machine.
`rosnode kill`	Kills a running node.

Examples:

Kill all nodes:
```
$ rosnode kill -a
```
List nodes on a machine:
```
$ rosnode machine aqy.local
```
Ping all nodes:
```
$ rosnode ping --all
```

roslaunch

Starts ROS nodes locally and remotely via SSH, as well as setting parameters on the parameter server.

Examples:

Launch on a different port:
```
$ roslaunch -p 1234 package filename.launch
```
Launch a file in a package:
```
$ roslaunch package filename.launch
```
Launch on the local nodes:
```
$ roslaunch --local package filename.launch
```

rostopic

A tool for displaying debug information about ROS topics, including publishers, subscribers, publishing rate, and messages.

Commands:

`rostopic bw`	Display bandwidth used by topic.
`rostopic echo`	Print messages to screen.
`rostopic hz`	Display publishing rate of topic.
`rostopic list`	Print information about active topics.
`rostopic pub`	Publish data to topic.
`rostopic type`	Print topic type.
`rostopic find`	Find topics by type.

Examples:

Publish hello at 10 Hz:
```
$ rostopic pub -r 10 /topic_name std_msgs/String hello
```
Clear the screen after each message is published:
```
$ rostopic echo -c /topic_name
```
Display messages that match a given Python expression:
```
$ rostopic echo --filter "m.data=='foo'" /topic_name
```
Pipe the output of rostopic to rosmsg to view the msg type:
```
$ rostopic type /topic_name | rosmsg show
```

rosparam

A tool for getting and setting ROS parameters on the parameter server using YAML-encoded files.

Commands:

`rosparam set`	Set a parameter.
`rosparam get`	Get a parameter.
`rosparam load`	Load parameters from a file.
`rosparam dump`	Dump parameters to a file.
`rosparam delete`	Delete a parameter.
`rosparam list`	List parameter names.

Examples:

List all the parameters in a namespace:
```
$ rosparam list /namespace
```
Setting a list with one as a string, integer, and float:
```
$ rosparam set /foo "['1', 1, 1.0]"
```
Dump only the parameters in a specific namespace to file:
```
$ rosparam dump dump.yaml /namespace
```

rosservice

A tool for listing and querying ROS services.

Commands:

`rosservice list`	Print information about active services.
`rosservice node`	Print the name of the node providing a service.
`rosservice call`	Call the service with the given args.
`rosservice args`	List the arguments of a service.
`rosservice type`	Print the service type.
`rosservice uri`	Print the service ROSRPC uri.
`rosservice find`	Find services by service type.

Examples:

Call a service from the command-line:
```
$ rosservice call /add_two_ints 1 2
```
Pipe the output of rosservice to rossrv to view the srv type:
```
$ rosservice type add_two_ints | rossrv show
```
Display all services of a particular type:
```
$ rosservice find rospy_tutorials/AddTwoInts
```

Logging Command-line Tools

rosbag

This is a set of tools for recording from and playing back to ROS topics. It is intended to be high performance and avoids deserialization and reserializationof the messages.

rosbag record will generate a ".bag" file (so named for historical reasons) with the contents of all topics that you pass to it.

Examples:
Record all topics:
```
$ rosbag record -a
```
Record select topics:
```
$ rosbag record topic1 topic2
```

rosbag play will take the contents of one or more bag file, and play them back in a time-synchronized fashion.

Examples:
Replay all messages without waiting:
```
$ rosbag play -a demo_log.bag
```
Replay several bag files at once:
```
$ rosbag play demo1.bag demo2.bag
```

Graphical Tools

rxgraph

Displays a graph of the ROS nodes that are currently running, as well as the ROS topics that connect them.

Usage:
```
$ rxgraph
```

rxplot

A tool for plotting data from one or more ROS topic fields using matplotlib.

Examples:
To graph the data in different plots:
```
$ rxplot /topic1/field1 /topic2/field2
```
To graph the data all on the same plot:
```
$ rxplot /topic1/field1,/topic2/field2
```
To graph multiple fields of a message:
```
$ rxplot /topic1/field1:field2:field3
```

rxbag

A tool for visualizing, inspecting, and replaying histories (bag files) of ROS messages.

Usage:
```
$ rxbag bag_file.bag
```

rxconsole

A tool for displaying and filtering messages published on rosout.

Usage:
```
$ rxconsole
```

tf Command-line Tools

tf_echo

A tool that prints the information about a particular transformation between a source_frame and a target_frame.

Usage:
```
$ rosrun tf tf_echo <source_frame> <target_frame>
```

Examples:
To echo the transform between /map and /odom:
```
$ rosrun tf tf_echo /map /odom
```

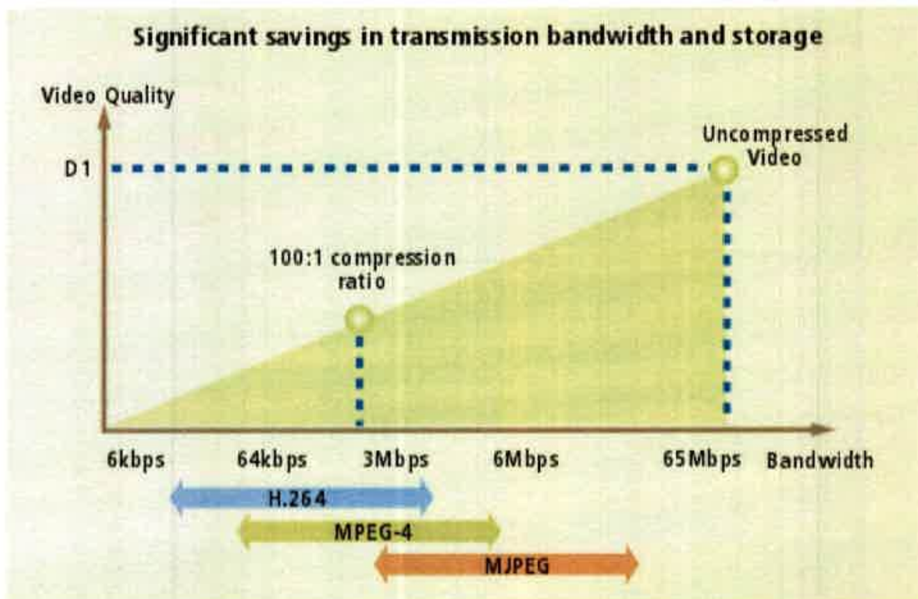
view_frames

A tool for visualizing the full tree of coordinate transforms.

Usage:
```
$ rosrun tf view_frames
$ evince frames.pdf
```

Copyright © 2010 Willow Garage

Anexo D: Comparativa entre formatos de video



FROM: http://www.marchnetworks.com/Documents/TechTip_09

Anexo E: Código en python de la clase *PID* implementada

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  #-----------------------------------------------------------------------------
4  # PID.py
5  # A simple implementation of a PID controller
6  #-----------------------------------------------------------------------------
7  # Example source code for the book "Real-World Instrumentation with Python"
8  # by J. M. Hughes, published by O'Reilly Media, December 2010,
9  # ISBN 978-0-596-80956-0.
10 #-----------------------------------------------------------------------------
11
12 import time
13
14 class PID:
15     """ Simple PID control.
16
17         This class implements a simplistic PID control algorithm. When first
18         instantiated all the gain variables are set to zero, so calling
19         the method GenOut will just return zero.
20     """
21     def __init__(self,p,i,d):
22         # initialze gains
23         self.Kp = p
24         self.Kd = d
25         self.Ki = i
26
27         self.Initialize()
28
29     def Initialize(self):
30         # initialize delta t variables
31         self.currtm = time.time()
32         self.prevtm = self.currtm
33
34         self.prev_err = 0
35
36         # term result variables
37         self.Cp = 0
38         self.Ci = 0
39         self.Cd = 0
40
41
42     def GenOut(self, error,p,i,d,Ci_min=-500000,Ci_max=500000,Out_min=0,Out_max=
       10000.65):
43         """ Performs a PID computation and returns a control value based on
44             the elapsed time (dt) and the error signal from a summing junction
45             (the error parameter).
46         """
47         self.Kp = p
48         self.Kd = d
49         self.Ki = i
50     self.Ci_min=Ci_min
51         self.Ci_max=Ci_max
52         self.Out_min=Out_min
53         self.Out_max=Out_max
54         self.currtm = time.time()              # get t
55     dt = self.currtm - self.prevtm          # get delta t
56         de = error - self.prev_err             # get delta error
```

```
57
58          self.Cp = self.Kp * error                  # proportional term
59          self.Ci += error * dt                      # integral term
60          #if self.Ci>self.Ci_max:                          #Acomodo limites
61          #    self.Ci=self.Ci_max
62          #elif self.Ci<self.Ci_min:
63          #    self.Ci=self.Ci_min
64
65          self.Cd = 0
66          if dt > 0:                                 # no div by zero
67              self.Cd = de/dt                        # derivative term
68
69          self.prevtm = self.currtm                  # save t for next pass
70          self.prev_err = error                      # save t-1 error
71
72          # sum the terms and return the result
73          self.Out=self.Cp + self.Ki * self.Ci + self.Kd * self.Cd
74          #if self.Out>self.Out_max:                          #Acomodo limites
75          #    self.Out=self.Out_max
76          #elif self.Out<self.Out_min:
77          #    self.Out=self.Out_min
78          return self.Out
79
80
```

Anexo F: Información sobre *netem*

netem

By Linux Foundatio... - November 19, 2009 - 10:23am

netem provides Network Emulation (http://en.wikipedia.com/wiki/Network_emulation) functionality for testing protocols by emulating the properties of wide area networks. The current version emulates variable delay, loss, duplication and re-ordering.

If you run a current 2.6 distribution, (Fedora (http://en.wikipedia.com/wiki/Fedora_Core), OpenSuse (http://en.wikipedia.com/wiki/Open_Suse), Gentoo (http://en.wikipedia.com/wiki/Gentoo_Linux), Debian (http://en.wikipedia.com/wiki/Debian), Mandriva (http://en.wikipedia.com/wiki/Mandriva), Ubuntu (http://en.wikipedia.com/wiki/Ubuntu_%28Linux_distribution%29)), then netem is already enabled in the kernel and a current version of iproute2 (/node/add/wiki?gids[]=5066) is included. The netem kernel component is enabled under:

```
Networking -->
  Networking Options -->
    QoS and/or fair queuing -->
       Network emulator
```

Netem is controlled by the command line tool 'tc' which is part of the iproute2 (/node/add/wiki?gids[]=5066) package of tools. The tc command uses shared libraries and data files in the /usr/lib/tc directory.

Contents

- 1 Examples (#Examples)
  - 1.1 Emulating wide area network delays (#Emulating_wide_area_network_delays)
  - 1.2 Delay distribution (#Delay_distribution)
  - 1.3 Packet loss (#Packet_loss)
    - 1.3.1 Caveats (#Caveats)
  - 1.4 Packet duplication (#Packet_duplication)
  - 1.5 Packet corruption (#Packet_corruption)
  - 1.6 Packet re-ordering (#Packet_re-ordering)
    - 1.6.1 Caveats (#Caveats_2)
  - 1.7 Rate control (#Rate_control)
  - 1.8 Non FIFO queuing (#Non_FIFO_queuing)
  - 1.9 Delaying only some traffic (#Delaying_only_some_traffic)
- 2 FAQ (#FAQ)
  - 2.1 How come first ping takes longer? (#How_come_first_ping_takes_longer.3F)
  - 2.2 How come TCP is so slow over netem? (#How_come_TCP_is_so_slow_over_netem.3F)
  - 2.3 How can I use netem on incoming traffic? (#How_can_I_use_netem_on_incoming_traffic.3F)
  - 2.4 How to reorder packets based on jitter? (#How_to_reorder_packets_based_on_jitter.3F)
  - 2.5 How does the value of HZ impact Netem? (#How_does_the_value_of_HZ_impact_Netem.3F)
- 3 Links (#Links)
- 4 Contact Info (#Contact_Info)

Examples

Emulating wide area network delays

This is the simplest example, it just adds a fixed amount of delay to all packets going out of the local Ethernet.

```
# tc qdisc add dev eth0 root netem delay 100ms
```

Now a simple ping test to host on the local network should show an increase of 100 milliseconds. The delay is limited by the clock resolution of the kernel (HZ). On most 2.4 systems, the system clock runs at 100hz which allows delays in increments of 10ms. On 2.6, the value is a configuration parameter from 1000 to 100 hz.

Later examples just change parameters without reloading the qdisc

Real wide area networks show variability so it is possible to add random variation.

```
# tc qdisc change dev eth0 root netem delay 100ms 10ms
```

This causes the added delay to be 100ms ± 10ms. Network delay variation isn't purely random, so to emulate that there is a correlation (http://en.wikipedia.com/wiki/correlation) value as well.

```
# tc qdisc change dev eth0 root netem delay 100ms 10ms 25%
```

This causes the added delay to be 100ms ± 10ms with the next random element depending 25% on the last one. This isn't true statistical correlation (http://en.wikipedia.com/wiki/correlation) , but an approximation.

Delay distribution

Typically, the delay in a network is not uniform. It is more common to use a something like a normal distribution (http://en.wikipedia.com/wiki/Normal_Distribution) to describe the variation in delay. The netem discipline can take a table to specify a non-uniform distribution.

```
# tc qdisc change dev eth0 root netem delay 100ms 20ms distribution normal
```

The actual tables (normal, pareto, paretonormal) are generated as part of the iproute2 (/node/add/wiki?gids[]=5066) compilation and placed in /usr/lib/tc; so it is possible with some effort to make your own distribution based on experimental data.

Packet loss

Random packet loss is specified in the 'tc' command in percent. The smallest possible non-zero value is:

232 = 0.0000000232%

```
# tc qdisc change dev eth0 root netem loss 0.1%
```

This causes 1/10th of a percent (i.e 1 out of 1000) packets to be randomly dropped.

An optional correlation may also be added. This causes the random number generator to be less random and can be used to emulate packet burst losses.

```
# tc qdisc change dev eth0 root netem loss 0.3% 25%
```

This will cause 0.3% of packets to be lost, and each successive probability depends by a quarter on the last one.

Probn = .25 * Probn-1 + .75 * Random

Caveats

- When loss is used locally (not on a bridge or router), the loss is reported to the upper level protocols. This may cause TCP to resend and behave as if there was no loss. When testing protocol reponse to loss it is best to use a netem on a bridge (/node/add/wiki?gids[]=5066) or router (/node/add/wiki?gids[]=5066)

Packet duplication

Packet duplication is specified the same way as packet loss.

```
# tc qdisc change dev eth0 root netem duplicate 1%
```

Packet corruption

Random noise can be emulated (in 2.6.16 or later) with the corrupt option. This introduces a single bit error at a random offset in the packet.

```
# tc qdisc change dev eth0 root netem corrupt 0.1%
```

Packet re-ordering

There are two different ways to specify reordering. The first method gap uses a fixed sequence and reorders every Nth packet. A simple usage of this is:

```
# tc qdisc change dev eth0 root netem gap 5 delay 10ms
```

This causes every 5th (10th, 15th, ...) packet to go to be sent immediately and every other packet to be delayed by 10ms. This is predictable and useful for base protocol testing like reassembly.

The second form reorder of re-ordering is more like real life. It causes a certain percentage of the packets to get mis-ordered.

```
# tc qdisc change dev eth0 root netem delay 10ms reorder 25% 50%
```

In this example, 25% of packets (with a correlation of 50%) will get sent immediately, others will be delayed by 10ms.

Newer versions of netem will also re-order packets if the random delay values are out of order. The following will cause some reordering:

```
# tc qdisc change dev eth0 root netem delay 100ms 75ms
```

If the first packet gets a random delay of 100ms (100ms base - 0ms jitter) and the second packet is sent 1ms later and gets a delay of 50ms (100ms base - 50ms jitter); the second packet will be sent first. This is because the queue discipline tfifo inside netem, keeps packets in order by time to send.

Caveats

- Mixing forms of reordering may lead to unexpected results
- Any method of reordering to work, some delay is necessary.
- If the delay is less than the inter-packet arrival time then no reordering will be seen.

Rate control

There is no rate control built-in to the netem discipline, instead use one of the other disciplines that does do rate control. In this example, we use Token Bucket (http://en.wikipedia.com/wiki/Token_bucket) Filter (TBF) to limit output.

```
# tc qdisc add dev eth0 root handle 1:0 netem delay 100ms
# tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate 256kbit buffer 1600 limit 3000
# tc -s qdisc ls dev eth0
qdisc netem 1: limit 1000 delay 100.0ms
 Sent 0 bytes 0 pkts (dropped 0, overlimits 0 )
qdisc tbf 10: rate 256Kbit burst 1599b lat 26.6ms
 Sent 0 bytes 0 pkts (dropped 0, overlimits 0 )
```

Check on the options for buffer and limit as you might find you need bigger defaults than these (they are in bytes)

For more explanation about how to use classful queuing disciplines see: Linux Advanced Routing HOWTO - classes (http://lartc.org/howto/lartc.qdisc.classful.html)

Non FIFO queuing

Just like the previous example, any of the other queuing disciplines (GRED, CBQ, etc) can be used.

Delaying only some traffic

Here is a simple example that only controls traffic to one IP address.

```
 # tc qdisc add dev eth0 root handle 1: prio
 # tc qdisc add dev eth0 parent 1:3 handle 30: \
tbf rate 20kbit buffer 1600 limit  3000
 # tc qdisc add dev eth0 parent 30:1 handle 31: \
netem  delay 200ms 10ms distribution normal
 # tc filter add dev eth0 protocol ip parent 1:0 prio 3 u32 \
     match ip dst 65.172.181.4/32 flowid 1:3
```

The commands makes a simple priority queueing discipline, then a TBF is added to do rate control, then attaches a basic netem. Finally, a filter classifies all packets going to 65.172.181.4 as being priority 3. For more info on traffic classification see LARTC -- filters (http://lartc.org/howto/lartc.qdisc.filters.html)

FAQ

How come first ping takes longer?

The first ICMP packet in a ping requires an ARP request/response as well.

How come TCP is so slow over netem?

When you run TCP (http://en.wikipedia.com/wiki/Transmission_Control_Protocol) over large Bandwidth Delay Product (http://en.wikipedia.com/wiki/Bandwidth-delay_product) links, you need to do some TCP tuning (http://en.wikipedia.com/wiki/TCP_Tuning) to increase the maximum possible buffer space.

How can I use netem on incoming traffic?

You need to use the Intermediate Functional Block pseudo-device IFB (/collaborate/workgroups/networking/ifb) . This network device allows attaching queuing discplines to incoming packets.

```
# modprobe ifb
# ip link set dev ifb0 up
# tc qdisc add dev eth0 ingress
# tc filter add dev eth0 parent ffff: \
  protocol ip u32 match u32 0 0 flowid 1:1 action mirred egress redirect dev ifb0
# tc qdisc add dev ifb0 root netem delay 750ms
```

Another way is to use another machine as an Ethernet bridge (/node/add/wiki?gids[]=5066) , and apply netem to both Ethernet devices.

How to reorder packets based on jitter?

Starting with version 1.1 (in 2.6.15), netem will reorder packets if the delay value has lots of jitter.

If you don't want this behaviour then replace the internal queue discipline tfifo with a pure packet fifo pfifo. The following example has lots of jitter, but the packets will stay in order.

```
# tc qdisc add dev eth0 root handle 1: netem delay 10ms 100ms
# tc qdisc add dev eth0 parent 1:1 pfifo limit 1000
```

How does the value of HZ impact Netem?

In the 2.6 line of kernels, HZ is a configurable parameter that takes values of either 100, 250, or 1000. Because it affects the granularity with which Netem is able to delay packets, it is most beneficial to set HZ to 1000, which will allow for delays in increments of 1ms. See this mailing list post (http://lists.osdl.org/pipermail/netem/2006-March/000343.html) for a more detailed discussion of the impact of HZ.

In kernel versions, 2.6.22 or later, netem will use high resolution timers, if they are enabled. This allows for finer granularity (sub-jiffie) resolution.

Links

- Linux Conf Au presentation (http://developer.osdl.org/shemminger/LCA2005_netem.pdf) and paper (http://developer.osdl.org/shemminger/LCA2005_paper.pdf) .
- dummynet an network emulator in FreeBSD (http://info.iet.unipi.it/~luigi/ip_dummynet/)
- NISTnet (http://snad.ncsl.nist.gov/itg/nistnet/)

Contact Info

Since netem is part of the core Linux subsystem, all bug reports and patches should be sent to Linux Network Developers (mailto:netdev@vger.kernel.org) mailing list.

The netem@osdl.org (mailto:netem@osdl.org) mailing list is available for user discussions. Mail from non-subscribers is moderated to prevent spam. To subscribe or unsubscribe use the Netem mailing list interface (http://lists.osdl.org/mailman/listinfo/netem) .

Groups:

Copyright © 2014 Linux Foundation. All rights reserved.
The Linux Foundation, LSB, Yocto Project, Tizen and IAccessible2 are registered trademarks of The Linux Foundation.
Linux Standard Base, LSB Certified, MeeGo, and the Linux Foundation Symbol are trademarks of The Linux Foundation.
Linux is a registered trademark of Linus Torvalds.
Please see our terms of use, antitrust policy, and privacy policy.

Anexo G: Definición de Clases –IP QoS (ITU-T)

Table 1/Y.1541 -- IP QoS Class Definitions and NP Objectives

Network Performance Parameter	Nature of Network Performance Objective	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5
IPTD	Upper bound on the mean IPTD	100 ms	400 ms	100 ms	400 ms	1 s	U
IPDV	Upper bound on the $1\text{-}10^{-3}$ quantile of IPTD minus the minimum IPTD	50 ms	50 ms	U	U	U	U
IPLR	Upper bound on the packet loss probability	$1*10^{-3}$	$1*10^{-3}$	$1*10^{-3}$	$1*10^{-3}$	$1*10^{-3}$	U
IPER	Upper bound	$1*10^{-4}$					U

From: http://www1.cs.columbia.edu/~hgs/papers/others/2000/2002/IP_QOS_Summary_Nov_2002.ppt

Anexo H: Capa de transporte *TCPROS*

TCPROS

TCPROS is a transport layer for ROS Messages (/Messages) and Services (/Services). It uses standard TCP/IP sockets for transporting message data. Inbound connections are received via a TCP Server Socket with a header containing message data type and routing information. For more information about this header format, see Connection Header (/ROS/Connection%20Header).

TODO: more information about wire protocol

1. TCPROS Header Fields

Inbound connections to a TCPROS Server Socket are routed by information contained within the header fields. If the header contains the 'topic' field, it will be routed as a connection to a ROS Publisher. If it contains a 'service' field, it will be routed as a connection to a ROS Service.

A TCPROS subscriber is required to send the following fields:

- message_definition: full text of message definition (output of gendeps --cat)
- callerid: name of subscriber
- topic: name of the topic the subscriber is connecting to
- md5sum: md5sum of the message type
- type: message type

A TCPROS publisher is required to reply with the following fields on a successful connection:

- md5sum: md5sum of the message type
- type: message type

A TCPROS service client is required to send the following fields:

- callerid: node name of service client
- service: name of the topic the subscriber is connecting to
- md5sum: md5sum of the message type
- type: service type

A TCPROS subscriber may optionally send the following fields:

- tcp_nodelay: if '1', indicates that the publisher should set TCP_NODELAY on the socket, if possible.

A TCPROS publisher may optionally send the following fields:

- callerid: name of publisher. Although this field is not required, it is highly recommended for debugging purposes.
- latching: if '1', indicates that the publisher is sending latched messages. The protocol for exchanging latched messages is identical, but subscribers may wish to take note of the latched status.

A TCPROS service client may optionally send the following fields:

- persistent: if '1', indicates that the service connection should be kept open for multiple service requests

A TCPROS Service is required to reply with the following fields on a successful connection:

- callerid: node name of Service

TCPROS has the following optional fields:

- error: human-readable error message if the connection is not successful

2. Service-specific

Services include an 'ok' byte in response to each service request message.

If the ok byte is true (1), it must be followed by the service response message.

If the ok byte is false (0), it must be followed by a serialized string representing the error message (same length + bytes format that ROS messages use for serializing strings, potentially the string can be empty which is the case if a service just returns false).

Except where otherwise noted, the ROS wiki is licensed under the Creative Commons Attribution 3.0 (http://creativecommons.org/licenses/by/3.0/) | Find us on Google+ (https://plus.google.com/113789706402978299308)

Wiki: ROS/TCPROS (last edited 2013-04-15 22:10:40 by unknownentity_1 (/unknownentity_1))

(http://www.osrfoundation.org)