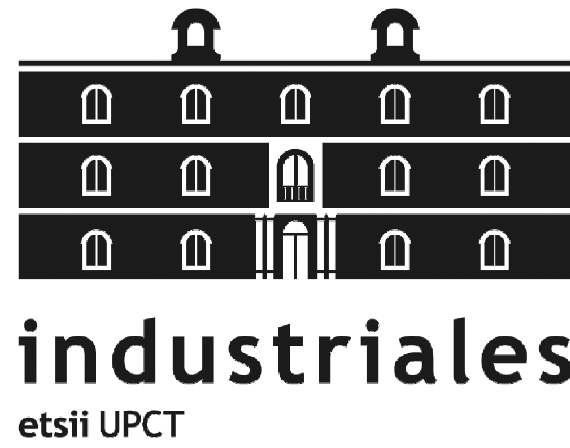


Programación de un robot Pioneer utilizando el framework SmartSoft

Titulación: Ingeniería en Automática y Electrónica Industrial

Alumno/a: José Ardil González
Directores: Dr. D. Diego Alonso Cáceres
Dr. D. Francisco J. Ortiz Zaragoza

Cartagena, 30 de Enero de 2012

A mis directores por darme la oportunidad de realizar este proyecto orientándome
en la realización del mismo.
A mi familia y compañeros sin los cuales estoy seguro no habría sido posible llegar hasta aquí.
A Ángela por su apoyo incondicional.

ÍNDICE

1. INTRODUCCIÓN Y OBJETIVOS **7**
1.1. Introducción y motivación 7
1.2. Objetivos del PFC 8
1.3. Estructura del documento 8

2. ESTADO DE LA TÉCNICA **11**
2.1. Ingeniería del software para robótica 11
2.2. Arquitecturas robóticas 13
2.3. Clasificación arquitecturas robóticas 16
2.3.1. Arquitecturas jerárquicas 17
2.3.2. Arquitecturas reactivas 21
2.3.3. Arquitecturas híbridas 25
2.4. Subsistemas y componentes típicos para una arquitectura híbrida 29
2.5. Aproximaciones reutilizables para programar aplicaciones robóticas 31
2.5.1. Frameworks OO 32
2.5.2. Frameworks OC 41
2.5.3. MDE 50

3. SMARTSOFT **55**
3.1. Introducción 55
3.2. SmartSoft MDSD Toolchain 57
3.3. Elementos fundamentales de SmartSoft 59
3.3.1. Objetos de comunicación 59
3.3.2. Componentes 64
3.3.3. Puertos de comunicación 69
3.3.4. Deployment 71
3.4. Patrones de comunicación 72
3.4.1. Patrón de comunicación Send 72
3.4.2. Patrón de comunicación Query 74
3.4.3. Patrón de comunicación PushNewest 76
3.4.4. Patrón de comunicación PushTimed 78
3.4.5. Patrón de comunicación Event 80
3.4.6. Patrón de comunicación Wiring 82
3.4.7. Patrón de comunicación State 83

4. PRIMEROS PASOS CON SMARTSOFT **85**
4.1. Puesta en marcha de SmartSoft 85
4.2. Descripción de la herramienta de desarrollo 85
4.3. Descripción del proceso de diseño utilizando el toolchain 89
4.3.1. Construcción de objetos de comunicación 89
4.3.2. Construcción de componentes 92
4.3.3. Construcción de deployment 96
4.3.4. Ejecución de la aplicación 98
4.4. Estructura de directorios de SmartSoft 98

5. EJEMPLOS BÁSICOS **101**
5.1. Ejemplo de funcionamiento patrón Send 101
5.2. Ejemplo de funcionamiento patrón Query 107
5.3. Ejemplo de funcionamiento patrón PushNewest 113
5.4. Ejemplo de funcionamiento patrón PushTimed 119

6. DESARROLLO COMPLETO DE UNA APLICACIÓN UTILIZANDO SMARTSOFT **127**
6.1. Desarrollo de una aplicación robótica utilizando el simulador PlayerStage 127
6.1.1 Construcción del objeto de comunicación *CommObjectGoTo* 128
6.1.2. Copia del componente *SmartRobotConsole* 129
6.1.3. Análisis del componente *SmartPlayerSimulator* 134
6.1.4. Modificación del componente *SmartUPCTConsole* 135
6.1.5. Construcción del componente *ExSmartGoTo* 138
6.1.6. Construcción del deployment *DeployGoTo* 142
6.1.7. Ejecución de la aplicación 143
6.2. Desarrollo de una aplicación robótica utilizando el robot Pioneer 3-AT 145
6.2.1. Análisis del componente *SmartPioneerBaseServer* 146
6.2.2. Construcción del deployment *DeployGoToRobot* 148
6.2.3. Ejecución de la aplicación 149

7. CONCLUSIONES Y TRABAJOS FUTUROS **155**

8. BIBLIOGRAFÍA **159**

ANEXOS
Anexo A - Puesta en marcha de la Imagen VMWare 163
Anexo B - Instalación SmartSoft 165
Anexo C - Manual de usuario del robot Pioneer 3-AT 167

ÍNDICE DE FIGURAS

2.1. Robots Roomba de iRobot y MOSRO de Roboserv 11
2.2. Robots NAO de Alderbaran Robotics y Pioneer P3AT de ActivMedia 12
2.3. Primitivas robóticas 17
2.4. Primitivas en la arquitectura jerárquica 17
2.5. Idea de Rodney Brooks sobre el paradigma jerárquico 18
2.6. Esquema serie de una arquitectura jerárquica 18
2.7. Primitivas SENSE-PLAN-ACT en NASREM 19
2.8. Esquema general de la arquitectura de control NASREM. 20
2.9. Descomposición funcional de la arquitectura NASREM 20
2.10. Arquitectura reactiva 21
2.11. Esquema paralelo de una arquitectura reactiva. 21
2.12. Descomposición vertical de tareas (comportamientos) asociada al paradigma reactivo 22
2.13. Diagrama de bloques de la arquitectura Subsumption 23
2.14. Máquina de estado finito en la arquitectura Subsumption 24
2.15. Esquema de los tres primeros niveles de la arquitectura 24
2.16. Organización del paradigma híbrido 25
2.17. Combinación serie y paralelo de módulos en una arquitectura híbrida 26
2.18. Arquitectura Aura 27
2.19. Relación entre los diferentes esquemas de la arquitectura Aura 27
2.20. Planning utilizando campos potenciales 28
2.21. Subsistemas típicos en una arquitectura híbrida para robots móviles 29
2.22. Componentes típicos de cada subsistema de una arquitectura híbrida 29
2.23. Utilización de una librería 23
2.24. Utilización de un framework 34
2.25. ACE y sus componentes 35
2.26. Capa de comunicación entre componentes (middleware) 35
2.27. OMG modelo de referencia de la arquitectura 36
2.28. Arquitectura CORBA ORB 36
2.29. The ACE ORB (TAO) 37
2.30. Servidor de dispositivos internos de Player 38
2.31. Capas de abstracción de MIRO 39
2.32. Patrones de acción organizados en una jerarquía de comportamientos 39
2.33. Pequeño subgrupo de la API orientada a objetos de Webots 40
2.34. API Webots 40
2.35. Reutilización de componentes electrónicos en el diseño de un dispositivo 41
2.36. Hipotético sistema de reserva de vacaciones representado en UML 42
2.37. Componente OROCOS 45
2.38. Ejemplo de sistema en Orca 46
2.39. Niveles de abstracción de Marie 47
2.40. Unidad de control centralizada de Marie 47
2.41. Diagrama sencillo VPL para conducir un robot: 49
2.42. Programación tradicional frente al desarrollo dirigido por modelos 50
2.43. MDA en el contexto de desarrollo de software robótico 51
2.44. Transformaciones M2M y M2T 51
2.45. Editor de diagramas WebML 52
2.46. Vista general del proceso de desarrollo con V3Studio 53

3.1. Enfoque general SmartSoft. 55
3.2. Desarrollo de ACE/SmartSoft, Corba/SmartSoft y SmartSoft/MDSD 56
3.3. Eclipse Modeling Project 57
3.4. Proceso de diseño con SmartSoftMDSD 58
3.5. Objetos de comunicación 59
3.6. Objeto de comunicación CommSampleTime 59
3.7. ACE_CDR data types 60

3.8. Estructura de datos del objeto 61
3.9. Interfaz del framework 62
3.10. Objeto de comunicación completo 63
3.11. Comunicación entre componentes SmartSoft 64
3.12. Componente SmartSoft 65
3.13. Modelo de una tarea SmartTask 66
3.14. Modelo de un Handler 66
3.15. Modelo de un SmartMutex 66
3.16. Modelo de un SmartTimer 66
3.17. Modelo de un SmartIniParameterGroup 67
3.18. Ejemplo de Componente en SmartSoft 67
3.19. Conjunto de patrones de comunicación de SmartSoft 69
3.20. Distintas API's para los puertos 70
3.21. Ejemplo de Deployment para una aplicación de navegación 71
3.22. Patrón Wiring 82
3.23. Wiring slaves 83
3.24. State Master/Slave 83
3.25. Patrón State 84

4.1. Arquitectura típica de una máquina virtual. 85
4.2. Selección del espacio de trabajo 86
4.3. MDSD SmartSoft trabajando con un componente. 88
4.4. MDSD SmartSoft trabajando con un deployment 88
4.5. Aspecto de un proyecto del tipo repositorio de objetos de comunicación 90
4.6. Construcción de un objeto de comunicación mediante el toolchain 91
4.7. Ficheros generados con el proceso Run Code Generator en un objeto 91
4.8. Apartado SmartSoft Component de la vista Palette 93
4.9. Construcción de un componente mediante el toolchain 94
4.10. Ficheros generados con el proceso Run Code Generator en un componente. 95
4.11. Construcción de un deployment mediante el toolchain 97

5.1. Diagrama de funcionamiento ejemplo Send 101
5.2. Modelado del componente SampleSendServer 102
5.3. Modelado del componente SmartSendClient 104
5.4. Modelado del Deployment DeploySampleSend 105
5.5. Ejecución de DeploySampleSend 106
5.6. Diagrama de funcionamiento ejemplo Query 107
5.7. Modelado del componente SampleQueryServer 108
5.8. Modelado del componente SmartQueryClient 110
5.9. Modelado del Deployment DeploySampleQuery 111
5.10. Ejecución de DeploySampleQuery 112
5.11. Diagrama de funcionamiento ejemplo PushNewest 113
5.12. Modelado del componente SmartPushNewestServer 114
5.13. Modelado del componente SmartPushNewestClient 116
5.14. Modelado del Deployment DeploySamplePushNewest 117
5.15. Ejecución de DeploySamplePushNewest 118
5.16. Diagrama de funcionamiento ejemplo PushTimed 119
5.17. Modelado del componente SamplePushTimedServer 120
5.18. Modelado del componente SamplePushTimedClient 122
5.19. Modelado del deployment DeploySamplePushTimed 124
5.20. Ejecución de DeploySamplePushTimed 124

6.1. Esquema general de la aplicación 127
6.2. Modelado del objeto CommObjectGoTo 128
6.3. Importar proyecto al workspace 133
6.4. Componente SmartUPCTConsole 133
6.5. Modelo del componente SmartPlayerSimulator 134

6.6. Puerto de comunicación navigationVelocitySendServer 135
6.7. Puerto de comunicación basePositionPushTimedServer 135
6.8. Modelo original del componente SmartUPCTConsole 136
6.9. Puerto de comunicación positionToGoalClient 136
6.10. Puerto de comunicación basePositionPushTimed 137
6.11. Modelo del componente ExSmartGoTo 138
6.12. Puerto de comunicación positionToGoalServer 139
6.13. Puerto de comunicación basePositionPushTimedClient 139
6.14. Puerto de comunicación navigationVelocitySendClient 140
6.15. Modelo del deployment DeployGoTo 143
6.16. Ejecución de la aplicación utilizando el simulador PlayerStage - 1 144
6.17. Ejecución de la aplicación utilizando el simulador PlayerStage - 2 144
6.18. Ejecución de la aplicación utilizando el simulador PlayerStage - 3 145
6.19. Esquema general de la aplicación utilizando el robot Pioneer 146
6.20. Modelo del componente SmartPioneerBaseServer 147
6.21. Puerto de comunicación navigationVelocityHandler 147
6.22. Puerto de comunicación basePositionServer 148
6.23. Modelo del deployment DeployGoToRobot 149
6.24. Ejecución de la aplicación en el robot Pioneer 3-AT - 1 152
6.25. Ejecución de la aplicación en el robot Pioneer 3-AT - 2 152
6.26. Ejecución de la aplicación en el robot Pioneer 3-AT - 3 153
6.27. Movimiento del robot Pioneer 3-AT por el laboratorio - 1 153
6.28. Movimiento del robot Pioneer 3-AT por el laboratorio - 2 154
6.29. Movimiento del robot Pioneer 3-AT por el laboratorio - 3 154

c.1. Robot Pioneer 3-AT 167
c.2. Familia de robots Pioneer 168
c.3. Entradas y salidas del robot Pioneer 3-AT del laboratorio 168
c.4. Interruptor general del robot 169
c.5. Panel de control con el empotrado encendido 169
c.6. Consola en robot Pioneer 3-AT 170

1. INTRODUCCIÓN Y OBJETIVOS

1.1. Introducción y motivación

Cada día se dan importantes pasos en prototipos de laboratorio, consiguiendo robots más fiables y robustos. Las funciones vitales de los robots son proporcionados por el software, cuyo dominio sigue creciendo. Durante mucho tiempo se ha considerado que la integración requiere únicamente un esfuerzo menor, una vez que los algoritmos necesarios están disponibles. Sin embargo, con bastante frecuencia las dificultades a superar han sido enormemente subestimadas. Dominar la complejidad del software no sólo es una tarea exigente, sino también indispensable para conseguir que algún día los robots se puedan integrar libremente en entornos con humanos.

Los enfoques basados en componentes proporcionan los medios adecuados para dominar el problema de la complejidad mediante la división de un sistema complejo en varias unidades independientes con interfaces bien definidas. La interconexión de los componentes está garantizada por las normas impuestas por las interfaces de estos. Esto permite la creación de un sistema como la composición de distintos componentes testeados y comprobados. De esta manera, el constructor de componentes se centrará en el componente que esté desarrollando en concreto sin preocuparse de la estructura y detalles internos del resto de componentes con los que se comunicará. Sin embargo aún haciendo uso de la Ingeniería de Software basada en componentes (CBSE), la construcción de componentes para la robótica que sean compartidos, distribuidos y reutilizables sigue siendo un gran reto. Esto produce que algoritmos que son conocidos y siempre utilizados en las distintas aplicaciones, son implementados una y otra vez desde cero dando lugar a la pérdida de valiosos recursos.

Una de las principales razones que origina este hecho, es la falta de un modelo software de componentes que tenga en cuenta las necesidades de la robótica. En el campo de la robótica, existen con distinto grado de exigencia muchos requisitos tanto funcionales como no funcionales. El reto de estos enfoques basados en componentes para sistemas robóticos, es ayudar en la construcción de un sistema que permita componer aplicaciones a partir de componentes estandarizados, y a la vez proporcionar una arquitectura software sin la imposición de cierta arquitectura de robot en particular. Entre las distintas aproximaciones del estado de la técnica, SmartSoft ofrece un prometedor modelo de componentes.

SmartSoft es una plataforma de desarrollo para software robótico basada en un conjunto de patrones de comunicación como base del modelo de componentes robóticos. El framework ayuda al desarrollador de componentes, al generador de aplicaciones y al usuario final en la construcción y el uso de componentes, distribuidos de tal manera que la semántica de la interfaz de los componentes está predefinida por los patrones de comunicación, con independencia del lugar en que se aplican.

Puesto que todas las interfaces visibles desde el exterior que poseen los componentes se componen de los mismos patrones de comunicación, podemos decir que estos son la clave para conseguir una semántica estricta en cuanto a las comunicaciones entre componentes. Esto permite implementar sistemas débilmente acoplados y distribuidos basados en componentes estándar, cuya interacción puede ser ajustada de acuerdo al contexto y necesidades que se den en cada caso, así como garantizar la componibilidad mediante la restricción en la diversidad de estas interfaces. Además, el cableado dinámico de los componentes en tiempo de ejecución es apoyado explícitamente por un patrón independiente que interactúa fuertemente con las primitivas de comunicación. Esto produce una de las principales diferencias con otros enfoques.

El framework SmartSoft como una aplicación construida haciendo uso de este enfoque, ya ha demostrado su idoneidad en muchos proyectos. Aunque SmartSoft se ha desarrollado en el ámbito de los sistemas robóticos, no está restringido únicamente a las aplicaciones de ese dominio.

Con el objetivo de contrastar las ventajas que proporciona SmartSoft en el desarrollo de software para robots del DSIE comparado con otras aproximaciones, en este Proyecto Fin de Carrera (PFC en adelante) se realiza un estudio del enfoque adoptado por el equipo de desarrollo de SmartSoft a la hora de realizar una aplicación robótica completa. Los objetivos concretos del PFC aparecen el siguiente apartado.

1.2. Objetivos del PFC

Los objetivos del PFC, podemos dividirlos en 3 puntos:

- Primero se analizará el estado de la técnica en el campo de la Ingeniería de Software para la robótica. Se abordarán las distintas arquitecturas robóticas, así como las posibles formas con las que podemos abordar su implementación: programación modular, frameworks orientados a objetos, frameworks orientados a componentes y desarrollo de software dirigido por modelos. Finalmente acabaremos profundizando en el desarrollo de software basado en componentes.

- Una vez descrita la tecnología nos centraremos en el framework SmartSoft analizando en detalle su funcionamiento. Además de la descripción minuciosa de esta aplicación, se realizará un estudio de los beneficios que se obtienen haciendo uso de este software, así como por qué lo hacen más recomendable a otras opciones.
  Analizaremos el toolchain describiendo cada una de las partes que lo forman. Veremos el uso que hacen de éste los usuarios según sean sus roles (constructor de objetos de comunicación, constructor de componentes o generador de aplicaciones), y la manera en la que llevan a cabo su trabajo.

- Finalmente haremos uso de SmartSoft. En el desarrollo de software para robots del DSIE, iremos subiendo el nivel de complejidad de nuestros diseños, realizando desde pequeños ejemplos donde comprobar el funcionamiento de los distintos patrones de comunicación, hasta la implementación de una aplicación de navegación en el robot Pioneer P3AT por el laboratorio de DSIE (División de Sistemas e Ingeniería Electrónica) haciendo uso de este framework. También utilizaremos el simulador *Player Stage* en la ejecución de algunos diseños.

1.3. Estructura del documento

Además de este capítulo introductorio, este PFC se compone de 7 capítulos más (8 en total) y de varios anexos. En este apartado se describe el contenido de cada uno de ellos.

- En el capítulo 2 se hace un estudio sobre el estado de la técnica en el campo de la ingeniería software para la robótica. En este capítulo se analizan las distintas arquitecturas robóticas así como los módulos más importantes. A continuación se describirán las distintas posibilidades actuales que puede adoptar un usuario para llevar a cabo su implementación, centrándonos en el desarrollo de software haciendo uso de frameworks orientados a componentes. Por último, se analizan los requisitos funcionales y no funcionales así como los roles de usuarios, que existen en todo proceso de desarrollo software para la robótica basado en componentes.

- En el capítulo 3 se presenta el entorno de programación para aplicaciones robóticas que ofrece SmartSoft. Se describirá el modelo de componentes adoptado por el equipo de desarrollo de SmartSoft, así como sus elementos más importantes (en especial los patrones de comunicación). También se analiza el toolchain que SmartSoft proporciona, así como el framework que facilita el soporte de ejecución.

- En el capítulo 4 se describe el proceso de desarrollo software haciendo uso de SmartSoft. Se analizan todos los pasos necesarios para llevar a cabo la implementación y puesta en marcha de la aplicación robótica, desde los primeros pasos con la herramienta de desarrollo, la estructura de ficheros que utiliza SmartSoft, o la ejecución de un deployment.

- Una vez vistos todos los pasos necesarios para construir una aplicación haciendo uso de la herramienta de desarrollo de SmartSoft, en el capítulo 5 se muestran algunos ejemplos prácticos bastante sencillos.

Este capítulo también tiene como finalidad mostrar el funcionamiento de algunos los patrones anteriormente documentados.

- En el capítulo 6 se describe el proceso completo de construcción de una aplicación robótica para el robot Pioneer P3-AT. La aplicación se basa el movimiento hacia una “posición y ángulo objetivo” que el usuario introduce por teclado a la que el robot debe desplazarse de forma autónoma. Para ello utilizaremos tanto el simulador PlayerStage como el robot real Pioneer P3-AT que se desplazará por el laboratorio del DSIE.

- En el capítulo 7 se encuentran las conclusiones obtenidas tras la realización de este PFC. De la misma manera, se introducen algunos de los trabajos futuros que se llevarán a cabo por el DSIE siguiendo el enfoque adoptado por SmartSoft.

- En el capítulo 8 se encuentran las referencias de toda la bibliografía empleada en el PFC.

- Por último, en los anexos encontramos documentación sobre la puesta en marcha de SmartSoft utilizando la imagen VMWare proporcionada por el equipo de desarrollo, o bien siguiendo el proceso típico de instalación. En el último anexo encontramos un manual de usuario sobre el robot Pioneer P3-AT.

2. ESTADO DE LA TÉCNICA

2.1. Ingeniería del software para robótica

El comportamiento de un robot autónomo viene determinado por el programa que gobierna sus actuaciones. La creación de programas para robots debe cumplir ciertos requisitos específicos frente a la programación en otros entornos más tradicionales, como el ordenador personal. En los últimos años han surgido con fuerza plataformas de desarrollo, con la idea de facilitar la construcción incremental de estas aplicaciones robóticas. Más allá del acceso básico a los sensores y actuadores, las plataformas suelen proporcionar un modelo para la organización del código y bibliotecas con funcionalidades comunes.

La existencia de robots que realicen autónomamente tareas de modo eficiente depende fundamentalmente de su construcción mecánica y de su programación. Una vez construido el cuerpo mecánico del robot, conseguir que realice una tarea se convierte en la práctica en un problema de programación. La generación del comportamiento en un robot consiste entonces en escribir el programa que al ejecutarse en el robot causa ese comportamiento cuando este se encuentra en cierto entorno. La autonomía y la "inteligencia" residen en ese programa. En la robótica móvil el comportamiento principal es su movimiento. Los programas que se ejecutan en el robot determinan como se mueve este por el entorno, reaccionando ante obstáculos percibidos por los sensores, acercándose a algún destino, etc. y para ello tienen que enviar continuamente las órdenes pertinentes a los motores.

Hoy en día empiezan a comercializarse cada vez más robots que proporcionan servicios muy básicos. La robótica de servicio se centra en el diseño y construcción de máquinas capaces de proporcionar servicios directamente a la sociedad. Estos robots se venden como productos cerrados, programados por el fabricante e inalterables. Un ejemplo de este tipo de robot móvil es la aspiradora robótica *Roomba* de *iRobot*, o el robot de vigilancia *MOSRO* de *Roboserv.*

Figura 2.1: Robots Roomba de iRobot y MOSRO de Roboserv

En contraste, otra parte relevante del mercado son los robots programables. Los principales clientes de estos robots son los centros de investigación, que normalmente están interesados en programarlos ellos mismos. En estos casos, el fabricante vende los robots con un software que permite su programación por parte del usuario. Ejemplos de robots programables son el pequeño robot humanoide *Nao* desarrollado por la empresa francesa *Aldebaran Robotics*, que sustituye al perro robot *Aibo* de *Sony* como la plataforma estándar en la *Robocup*; o el robot *Pioneer P3-AT* de *ActivMedia* como del que dispone el DSIE.

Figura 2.2: Robots NAO de Alderbaran Robotics y Pioneer P3AT de ActivMedia

Para crear aplicaciones en el caso del Pioneer, *ActivMedia* proporciona el software *Advanced Robotics Interface for Applications* (*ARIA*), el cual incluye *ARNetworking*. *ARIA* es un entorno de desarrollo de código abierto basado en C++ que proporciona una interfaz robusta de cliente a una variedad de sistemas robóticos inteligentes. *ARNetworking* proporciona la capa crítica para las comunicaciones basadas en TCP/IP con el robot. ARIA contiene comportamientos básicos como la navegación segura, evitando obstáculos. Sin embargo, no incluye funcionalidad común como la construcción de mapas o la localización, que se venden por separado.

En el manejo de robots conviene diferenciar entre el usuario final y el programador, pues el robot ofrece interfaces muy distintas a uno y a otro. El usuario es quién aprovecha las aplicaciones que ha desarrollado el programador. Para él, la interfaz de uso deber ser extremadamente sencilla, máxime si el usuario es el público en general. Por ejemplo, la interfaz de uso de la aspiradora *Roomba* es un simple botón. Por el contrario, la interfaz de programación requiere de ciertos conocimientos por parte del desarrollador, y su objetivo es permitir la creación de programas para el robot. En este proyecto, se analizarán las diferentes opciones de abordar la compleja tarea de programación de robots móviles.

En los últimos años los avances en aplicaciones de robots móviles autónomos han sido significativos. Hoy en día, hay resultados fiables en algoritmos como la construcción de mapas, la navegación automática, la localización, la detección de caras o el procesamiento de visión artificial. Por ejemplo, en algunas fábricas hay robots que se mueven solos de un punto a otro remoto sin chocar con ningún obstáculo transportando piezas pesadas.

La funcionalidad de los prototipos de investigación construidos es cada vez mayor: guías de museos, cuidadores de ancianos, etc. No hay magia detrás de esas aplicaciones, hay programas que determinan el comportamiento de esos robots; programas que han escrito los diseñadores y que consiguen que el robot se mueva como lo hace.

No hay que confundir la metodología de la programación utilizada para conseguir comportamientos "inteligentes" en un robot de servicio, que normalmente suele ser autónomo, móvil y cuyo entorno de trabajo será desconocido (no-estructurados), con la programación de un robot industrial que se desenvuelve en un entorno generalmente estructurado. En los robots industriales los programas son simples secuencias estructuradas de reglas lógicas, aunque existen diferentes métodos de programación. Una posible clasificación es:

Programación por aprendizaje: Se hace que el robot realice una vez la tarea, y se registran las configuraciones adoptadas. Este aprendizaje puede ser: activo si el robot se mueve con sus propios actuadores según órdenes del usuario desde el panel de programación, o pasivo si los actuadores del robot están sin energía y el operario aporta la energía para mover el robot.

Las ventajas de este tipo de programación son: el programador puede conocer en detalle la tarea sin tener que ser un experto en la programación de robots, es el método de programación más sencillo y fácil de aprender, y que no se producen errores de posicionamiento por mala calibración del robot o su entorno. Por otro lado, algunas de las desventajas son: pocos tipos y número de instrucciones, difícil programar estructuras condicionales e iterativas, difícil de depurar y modificar programas, no pueden ser programas largos, difícil de interaccionar con E/S, hay que parar la cadena de producción...

Programación textual: La tarea del robot se expresa mediante una serie de instrucciones escritas en un lenguaje formal. La programación textual puede ser enfocada de distintas maneras: desde un punto de vista del robot, donde las instrucciones hacen referencia directa a las acciones del robot; desde un punto de vista de los objetos, en el que las instrucciones hacen referencia al modo en que deben quedar los objetos manipulados por el robot; o desde un punto de vista de las tareas, donde las instrucciones hacen referencia a la tarea u objetivo final a conseguir.

Las ventajas de este tipo de programación son: se aprovechan al máximo las capacidades del robot, se pueden especificar tareas más complejas, se integran fácilmente señales de otros dispositivos, y cuentan con instrucciones de control del flujo de programa. Por otro lado, las desventajas son que es un método más complicado que la programación por aprendizaje y necesita de un experto en programación

El enfoque por aprendizaje es utilizado en brazos industriales robotizados donde la ejecución deseada es el recorrido por una secuencia de puntos, los cuales se pueden introducir en el robot simplemente conduciendo externamente al brazo por las posiciones deseadas.

Este PFC se centra sin embargo en la programación de robots móviles, analizando las distintas posibilidades existentes a la hora de su programación. Entre los programadores de robots móviles figuran los propios fabricantes (como S*ony, ActivMedia, iRobot,* etc.), algunas empresas dedicadas (como *Evolution Robotics*), y los centros de investigación. Gran parte del software existente hoy día para robots ha sido desarrollado por grupos de investigación, lo cual está en consonancia con la inmadurez ya comentada del mercado robótico. Afortunadamente ese software suele estar disponible libremente en Internet, reflejo del deseo de difusión del conocimiento en esos grupos.

Ante los comportamientos tan complejos que se espera de un robot dotado de cierta inteligencia, la estructura de la programación se hace fundamental para conseguir su ampliación, mantenimiento y reutilización de partes comunes. Para ello, un aspecto crítico a la hora de desarrollar sistemas software complejos es el diseño de su arquitectura, representada como un conjunto de elementos computacionales y de datos interconectados de una cierta manera.

2.2. Arquitecturas robóticas

Introducción

Entendemos por arquitectura de control, una estructura que integra diferentes módulos, desarrollando una acción conjunta, aunque cada uno ellos desarrollen una determinada actividad. En Robótica Móvil, la arquitectura no debe considerarse como una mera integración de módulos, sino como algo más importante.

La estructura de dicha arquitectura va a determinar el comportamiento y la ejecución del sistema de control. Existen diferentes tendencias en el desarrollo de estas estructuras. Cada una de esas tendencias recibe el nombre de filosofía de control. Los requisitos básicos de una arquitectura de control en robótica móvil son múltiples. Algunos deben estar presentes en toda arquitectura, sin embargo, otros dependerán del tipo de aplicación para el que la arquitectura ha sido diseñada.

A continuación se hace un repaso de los principales requisitos que en general puede tener una arquitectura de control:

- ***Adaptabilidad.***
  La arquitectura de control de un robot debe diseñarse para permitir a éste desarrollar una variedad de tareas definidas mediante diversos parámetros. El control del sistema para cada una de estas tareas difiere en mayor o menor medida. Sin embargo, tanto el hardware del sistema como el software debe ser el mismo. Para conseguir esta doble condición de unicidad y diversidad, el funcionamiento del software no debe ser invariable, debe venir definido en función de los parámetros que caracterizan la tarea a realizar. Por otra parte, dicho funcionamiento va a depender también del entorno. Debido a que éste puede evolucionar durante el desarrollo de una tarea, el sistema de control debe ser capaz de adaptarse, automáticamente, al nuevo entorno. Este requisito se denomina también programabilidad.

- ***Flexibilidad.***
  Este requisito también se denomina "capacidad de evolución". En las aplicaciones relacionadas con la robótica móvil se requieren cambios continuos en el diseño durante la fase de implementación y puesta a punto. Para ello es necesario el uso de estructuras flexibles que permitan la evolución del diseño en función del buen o mal funcionamiento de alguno o algunos de sus módulos. Además, esta capacidad es importante sobre todo en proyectos innovadores donde se pueden producir revisiones significativas que impliquen un cambio de tecnologías, equipos, etc.

- ***Modularidad.***
  El sistema debe estar dividido en pequeños subsistemas que puedan ser diseñados, implementados y depurados separadamente. Esta característica es esencial para conseguir un diseño incremental, así como para un correcto mantenimiento del sistema y la detección y corrección de fallos.

- ***Ampliabilidad.***
  Una arquitectura de control se diseña para un objetivo concreto (que puede ser más o menos amplio). Sin embargo, este objetivo puede cambiar y puede ser necesario el uso del sistema para otros fines. Para cubrir estas situaciones es preciso que el sistema de control sea fácilmente ampliable. El requisito de modularidad favorece este otro.

- ***Eficiencia.***
  Este requisito hace referencia tanto a la eficiencia en la realización de las tareas (tiempo de ejecución, precisión,…), como a los recursos empleados. Esta característica se consigue con la utilización de modelos concretos, tanto del robot como del entorno, y la utilización de mecanismos de optimización.

- ***Autonomía.***
  El sistema debe ser capaz de realizar sus tareas por él mismo; gestionando sus propios recursos y sin recurrir a la intervención de dispositivos exteriores. Para obtener una verdadera autonomía, el robot debe ser capaz, no sólo de hacer frente a los cambios del entorno, sino además, controlar su comportamiento en función de su estado interno. Esta característica, que en principio es muy deseable, en algunas aplicaciones puede resultar inadecuada. Por ello, una posibilidad que puede y debe contemplarse desde el principio debido a su influencia en el diseño de la arquitectura, es la disponibilidad de una capacidad dual que permita un funcionamiento autónomo y teleoperado, cada uno de ellos cuando la tarea o el entorno lo requiera.

- ***Reactividad.***
  Se trata de la capacidad por la cual el sistema es capaz de detectar un evento en el momento en que se produce y reaccionar ante él, siempre en concordancia con el entorno y la tarea que se está desarrollando. Es necesario destacar la diferencia entre la aparición imprevista de un evento previsto y la aparición de un evento imprevisto. La primera situación es tratada en la mayoría de las arquitecturas, aunque el número de situaciones que prevén suele ser bastante reducido e incluso algunas de ellas no consiguen solucionarlo de forma satisfactoria.
  La reactividad, además, debe ser programable y controlada. Esta apreciación se entiende claramente si tenemos en cuenta la necesaria capacidad que debe tener todo sistema de detener su funcionamiento si el robot choca contra un objeto. Sin embargo, si planteamos una tarea para que el robot empuje un determinado objeto, resulta obvio que la anterior capacidad reactiva debe anularse.

- ***Robustez.***
  La robustez de un sistema se puede definir como la habilidad de éste para manejar entradas erróneas, incertidumbres y fallos repentinos en el sistema. Esta característica es muy importante si se desea que el sistema opere en tiempo real en un entorno dinámico. Este concepto está directamente relacionado con otro importante: la fiabilidad. Ese segundo concepto está relacionado con la no dependencia de un único sistema o subsistema para todas sus acciones. En general esto se consigue introduciendo redundancias, tanto hardware como software, de forma que si una de ellas falla, el sistema puede seguir realizando sus tareas con el sistema duplicado.

- ***Capacidad de decisión.***
  En muchas aplicaciones, es necesario que el robot exhiba ciertas características de alto nivel que le permitan razonar en función de información abstracta.

- ***Facilidad de manejo y supervisión.***
  A pesar de la posible autonomía de un robot móvil, su funcionamiento debe ser definido, lanzado y supervisado por un operario. Para esta labor no es preciso que dicha persona sea un especialista en robótica móvil. Para permitir esto, es necesario que el sistema de control sea fácil de manejar y supervisar, lo que implica la existencia de una interface entre el operario y el sistema amigable y que monitorice toda la información importante relativa a dicho funcionamiento.
  Algunos de los requisitos anteriores pueden llegar a ser conflictivos entre sí. En general, en el diseño actual de arquitecturas de control se discute el compromiso entre la eficiencia (robots dirigidos por objetivos o planes) y la adaptabilidad y capacidad de reacción. Esta dualidad que puede situar a la arquitectura en cualquier punto entre ambos ha dado lugar a diversos tipos de arquitecturas que serán analizadas en este tema.

Uno de los condicionantes más importantes a la hora de diseñar una arquitectura robótica es el tipo de tarea que el robot debe realizar.

Tareas a realizar por un robot móvil.

Realmente, el número de tareas que debe realizar un robot móvil presenta una alta dependencia respecto del objetivo para el que ha sido diseñado el robot (y con él su sistema de control) y respecto del propio robot. A continuación se citan algunas de estas tareas, con el objetivo de mostrar una idea de su elevado número y de la complejidad que debe tener el sistema de control que permita su complimiento.

- ***Captar información del entorno.***
  Para satisfacer muchos de los requisitos anteriores, el sistema de control debe tener conocimiento de los valores que tienen los diferentes parámetros que caracterizan tanto al robot como a su entorno en cada instante (al menos en instantes regulares de tiempo).

- ***Pre-procesar esa información.***
  Muchos datos obtenidos por los subsistemas sensoriales del robot presentan una fuerte componente de ruido y están afectados por numerosos errores, tanto sistemáticos como no sistemáticos, de diversa precedencia. Para poder manejar de forma eficiente esta información es necesario realizar sobre ellos tareas de pre-procesamiento, filtrado y fusión sensorial.

- ***Modelar el entorno.***
  Como se ha comentado anteriormente, la eficiencia del sistema se consigue mediante el manejo de modelos precisos tanto del robot como de su entorno y la utilización de técnicas de optimización que hacen uso de esa información. Si el robot se mueve el entornos dinámicos y/o desconocidos (total o parcialmente), será necesario disponer de elementos que realicen un proceso de modelado del entorno del robot de forma continua.

- ***Calcular trayectorias.***
  Con el mismo objetivo anterior de conseguir una eficiencia en el comportamiento del sistema de control, el movimiento de este desde un punto (o configuración) a otro/a debe realizarse siguiendo

una trayectoria que garantice ciertas restricciones de forma óptima. Para ello será necesario disponer de módulos de alto nivel, que utilizando algoritmos de optimización determinen esas trayectorias deseadas.

- ***Evitar colisiones.***
  Tal y como se ha comentado, los módulos de cálculo de trayectorias trabajan a partir de modelos del entorno. Si el entorno en el que se mueve el robot es dinámico y/o desconocido, las trayectorias obtenidas no garantiza la ausencia de colisiones entre el robot y elementos del entorno que hayan cambiado de posición o que eran desconocidos. Por ello, será importante para garantizar la autonomía del sistema, disponer de elementos que analicen en tiempo real la posibilidad de dichas colisiones con el fin de tomar las medidas adecuadas para evitarlas.

- ***Responder ante una orden.***
  Aunque el sistema de control suele estas diseñado para la consecución de determinas misiones, éstas no siempre están definidas en el proceso de diseño de la arquitectura de control, sino que son transmitidas de forma global o parcialmente en diferentes instantes de tiempo. El sistema, por tanto, debe ser capaz de recibir estas órdenes en tiempo real y actuar (responder) a las mismas.

Todas estas son tan sólo un pequeño subconjunto del total de tareas que un robot puede realizar, aunque dan una visión clara de la cantidad de tareas que el sistema de control debe ser capaz de afrontar. En los siguientes apartados se describen las principales arquitecturas de control en robótica móvil.

2.3. Clasificación arquitecturas robóticas

No existe un paradigma o una arquitectura robótica concreta que sea siempre la correcta, sino que ciertas arquitecturas serán más adecuadas para un tipo de problemas que para otros (si bien es cierto algunas de ellas están cayendo en desuso).

Aplicar e implementar la arquitectura más idónea hará más fácil la resolución de problemas. Por lo tanto, conocer las distintas arquitecturas de control robóticas es clave para ser capaces de programar con éxito una aplicación. También resulta interesante desde un punto de vista histórico conocer las diferentes arquitecturas, examinando los problemas que presentaban cada una de ellas y por qué se generó el cambio de una arquitectura a otra.

Para Robin Murphy (2000), los paradigmas se pueden dividir en dos clasificaciones según su comportamiento:

- Por la relación entre las tres primitivas comúnmente aceptadas de la robótica: sensorizar, planificar, actuar **(SENSE, PLAN, ACT).** Las funciones de un robot se pueden dividir en tres categorías muy generales.

  - Si una función toma en la información de los sensores del robot y produce una salida (output) útil para otras funciones, la función pertenece a la categoría SENSE.

  - A las funciones que toman la información (ya sea de los sensores o de su propio conocimiento sobre el entorno) y producen una o más tareas a realizar por el robot (ir por el pasillo, gire a la izquierda, continuar 3 metros y parar…), pertenecen a la categoría de PLAN.

  - Las funciones que producen los comandos de salida a los motores actuadores pertenecen a la categoría ACT (gira 98° en el sentido de las agujas del reloj con una velocidad de giro de 0.2mps).

Primitiva	Entrada	Salida	Descripción
SENSE	información sensorial "cruda"	modelo del entorno	traduce información proveniente de los sensores en un modelo del entorno
PLAN	modelo del entorno + conocimiento previo	directivas	genera el plan de ejecución acorde a la misión a cumplir y el modelo del entorno
ACT	información sensorial y/o directivas	comandos para los actuadores	traduce el plan de ejecución en una secuencia apropiada de comandos para los actuadores

Figura 2.3: Primitivas robóticas

- Por la manera en que la información recibida de los sensores es procesada y distribuida a través del sistema: o cuanto afecta al humano, animal o máquina lo que recibe por los sensores. Es decir, la reacción que se ofrece cuando se recibe información del exterior por medio de los sensores.

Teniendo en cuenta esta división creada por Robin Murphy, existen tres paradigmas sobre los que se basan las arquitecturas actuales: ***paradigma jerárquico*** (deliberativo o planificado), ***paradigma reactivo,*** y aunque no es un paradigma nuevo propiamente dicho, el ***paradigma híbrido*** que como su propio nombre indica es una mezcla de los dos primeros paradigmas.

2.3.1. Arquitecturas jerárquicas (SPA)

Desde la década de los sesenta hasta la de los noventa, la investigación y desarrollo de sistemas de control para sistemas autónomos se realizó en base a una descomposición jerárquica de las funciones a realizar por el sistema. Estas arquitecturas, también llamadas arquitecturas jerárquicas o arquitecturas planificadas, siguen un patrón claro detección-planificación-actuación. Es decir, es una arquitectura en la que primero detecta el entorno, posteriormente decide (delibera) la acción a realizar y por último realiza la acción decidida.

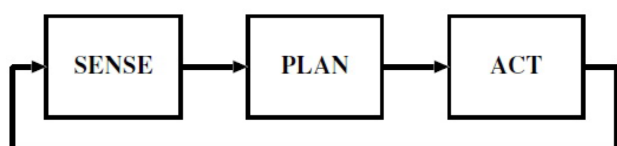


Figura 2.4: Primitivas en la arquitectura jerárquica

La arquitectura deliberativa se basa en enfoques tradicionales de la inteligencia artificial (IA) simbólica fundamentándose en la hipótesis de que el comportamiento inteligente puede ser logrado mediante un sistema de símbolos que permite la composición y tratamiento de los mismos que representan entidades del mundo real. Según Lope (2006), su filosofía podría ser resumida como "pensar mucho y luego actuar".

Para Barber Castaño (2000), estas arquitecturas se basan en dos fundamentos básicos:

- La descomposición funcional de la arquitectura: aporta la descomposición de la tarea en un refinamiento progresivo hasta que se reduzca al máximo el nivel de abstracción del problema, para tener más próxima la solución del mismo.

- La generación de un modelo de mundo: se genera este modelo de mundo en base a símbolos que son almacenados. A la hora de deliberar la acción a realizar, se consulta con los símbolos del mundo que se ha creado. Se podría presentar el problema de que si el mundo es cambiante mientras generamos estos símbolos, jamás vamos a tener un modelo de mundo fiel a la realidad, pero lo cierto es que este tipo de arquitecturas son utilizadas en mundos estáticos por lo que no presenta gran problema.

Se caracteriza por tener una representación interna central (RIC) con una descripción simbólica del ambiente que le rodea, las acciones del agente y sus objetivos. Además posee otros módulos como pueden ser planificación, aprendizaje, etc. que se comunican entre sí a través de RIC. Con lo que el RIC es el punto de paso para cualquier información recibida de los sensores. El razonamiento de la acción a ejecutar se realiza con un "*planning*" definido previamente el cual indicará al robot qué debe hacer para lograr alcanzar el objetivo. La ejecución de los planes, en su gran mayoría, es de lazo abierto, es decir, la salida no alimenta ninguna entrada.

Bajo este modelo se han distinguido dos niveles fundamentales: el nivel superior lleva a cabo tareas "inteligentes", en donde es preciso manejar datos abstractos y en donde se utilizan técnicas del ámbito de la Inteligencia Artificial (IA), presentando como principal inconveniente un alto tiempo de respuesta. Como contrapartida, el nivel inferior se caracteriza por trabajar con información concreta y con tiempos de respuesta bajos.

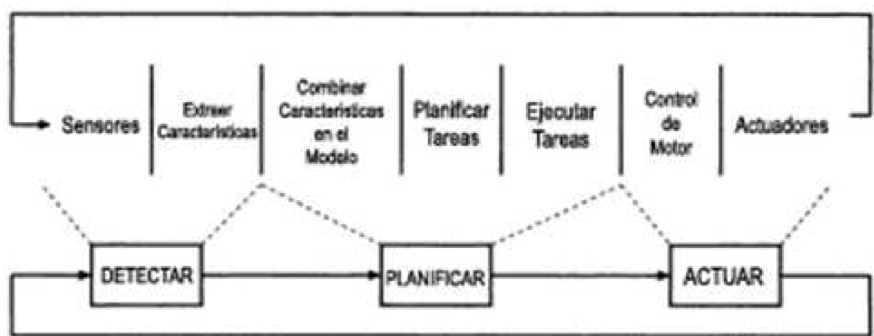


Figura 2.5: Idea de Rodney Brooks sobre el paradigma jerárquico

Habitualmente, en los sistemas desarrollados bajo este modelo jerárquico, las funciones localizadas en el nivel inferior son pocas y simples. En el caso, por ejemplo, de los robots móviles, éstos se encargan principalmente de gestionar la locomoción del vehículo y el control a bajo nivel de los sistemas sensoriales (por ejemplo, el posicionamiento de la torreta de visión).

La consecuencia principal que se deriva de la poca competencia del nivel inferior es que el sistema no reacciona con la rapidez necesaria a los cambios que pueden producirse en el entorno. En entornos dinámicos y/o parcialmente desconocidos, la necesidad de invocar a los niveles superiores es constante, y en un buen número de situaciones la intervención humana se vuelve imprescindible, con lo que el sistema pierde autonomía.

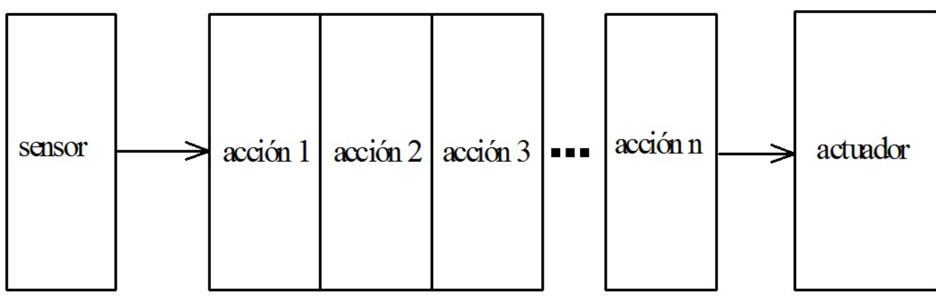


Figura 2.6: Esquema serie de una arquitectura jerárquica

Por otra parte, los diferentes módulos que intervienen en el sistema de control presentan una estructura de comunicación serie (figura 2.6), lo que genera una alta dependencia del comportamiento global del sistema respecto de la velocidad y la robustez de cada uno de los módulos que la conforman.

- Las ventajas de estas arquitecturas son que tenemos la habilidad para alcanzar objetivos de alto nivel de complejidad evitando errores y permite determinar planes de acción óptimos.

- Las desventajas mostradas por este tipo de arquitecturas son lentitud en el procesamiento de las tareas a realizar, ya que suelen estar basadas en técnicas de búsqueda muy pesadas y en ambientes no deterministas y cambiantes es necesario realizar añadidos para adaptarse a ese tipo de ambiente. De hecho, este tipo de arquitectura no suele ser utilizada cuando nos encontramos en un entorno cambiante.

Históricamente se han propuesto varias arquitecturas de control para lograr sistemas autónomos bajo la anterior filosofía de control. Un ejemplo de arquitectura deliberativa, jerárquica o planificada es la arquitectura **NASREM** (Nasa Standard Reference Model)**.**

Esta arquitectura creada por James S. Albus representó la culminación de 15 años de trabajo por parte del National Institute of Standards and Techonolgy (NIST) en la investigación de los sistemas en tiempo real para robots y máquinas inteligentes. En 1987 fue la primera vez que se utilizó esta arquitectura con un sistema en tiempo real. Esta arquitectura ha sido adoptada como sistema de control para manipuladores en las estaciones espaciales.

En esta arquitectura, el sistema de control es representado como tres niveles jerárquicos de módulos de computación ayudados por un sistema de comunicación y una memoria global.

- El primer modulo es la descomposición de tareas que planifica y ejecuta la descomposición de las metas iniciales en metas más pequeñas y manejables para el sistema. Este modulo implica simultáneamente una descomposición temporal y una descomposición espacial.

- El segundo módulo de la jerarquía es el modelado del mundo o entorno el cual se encarga de modelar y evaluar el entorno que rodea al sistema. El modelo de mundo es el mejor sistema de estimación y evaluación de la historia, estado actual y posibles estados futuros de nuestro entorno. Por supuesto, el conocimiento y mantenimiento de la información del entorno se realiza gracias a los sensores que obtienen la información y la introducen en el sistema.

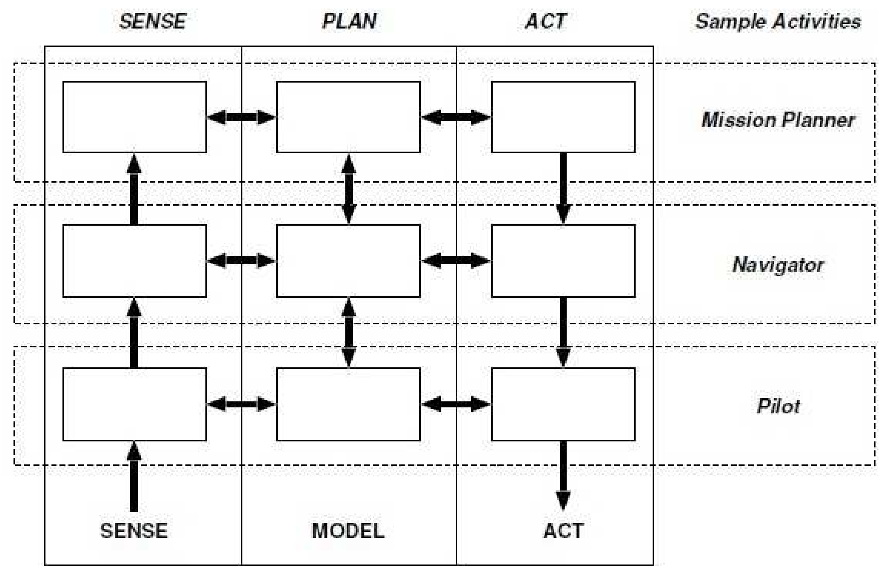


Figura 2.7: Primitivas SENSE-PLAN-ACT en NASREM

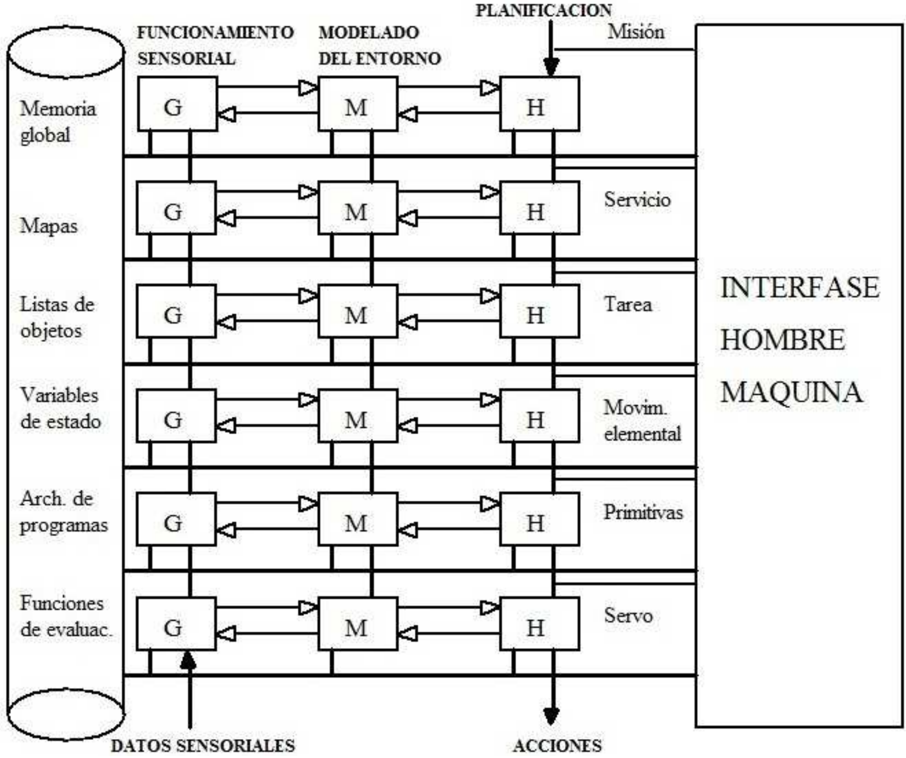


Figura 2.8: Esquema general de la arquitectura de control NASREM. Cada capa de control está estructurada convencionalmente con un procesamiento sensorial (G), un modelado del entorno (M) y procesos de supervisión y planificación de tareas (H).

- Y el último nivel de la jerarquía es el sistema de sensores. Se encarga de reconocer patrones, detectar eventos y filtrar e integrar la información del espacio y tiempo. Este modulo se encarga de comparar las predicciones hechas del sistema sobre el mundo real con las informaciones que se obtienen de los sensores y computa funciones de similitud y de diferencia. Por supuesto, este modulo se encarga de recoger los objetos, eventos y patrones que detecta en el entorno y enviarlos al modulo de modelización del entorno.

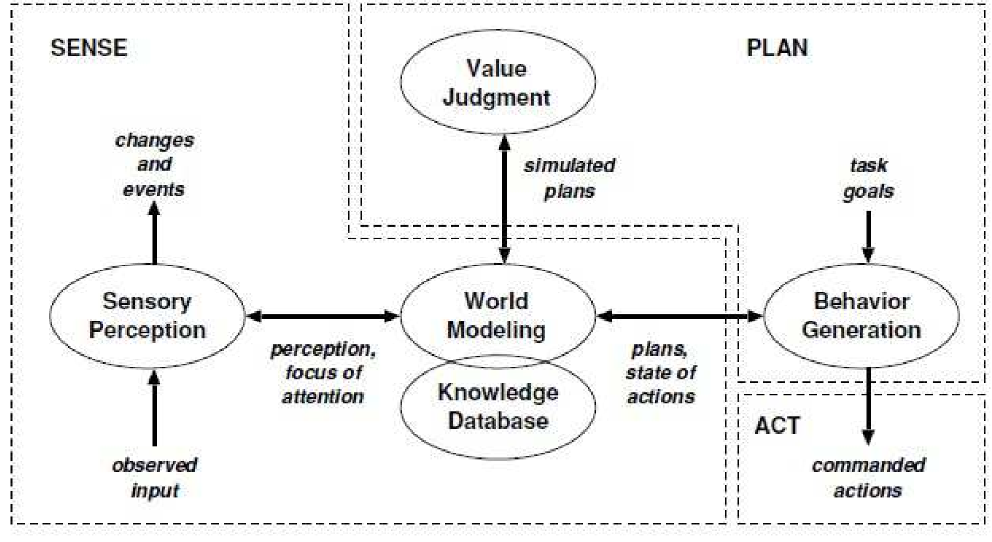


Figura 2.9: Descomposición funcional de la arquitectura NASREM

Otras implementaciones de arquitecturas clásicas son, por ejemplo:

- La desarrollada por Ford Motor Company desde 1982, y que se ha implementado en un robot móvil.
- Buttazo (1994) propone la implementación de una arquitectura jerárquica (Harems) que se basa en la transferencia de mensajes operando sobre un sistema operativo de tiempo real (Hartik), que se compone de los niveles de: aplicación, acción, comportamiento, comunicación y dispositivo.

2.3.2. Arquitecturas reactivas

La arquitectura reactiva, o paradigma reactivo, nació en los años ochenta para, posteriormente, ser el paradigma que siente las bases de otro paradigma, muy utilizado comúnmente, que es el hibrido. Además, los sistemas robóticos que tienen un dominio de tareas limitado todavía son construidos siguiendo este paradigma.

Para (Friedenberg, y otros, 2006), el paradigma reactivo "nace de la idea de que los comportamientos complejos surgen de los comportamientos simples y que operan de manera concurrente". Por otro lado este paradigma nace del desencanto con las arquitecturas deliberativas en cuanto a capacidad de tratar con la complejidad del mundo real y la imposibilidad de actuar en tiempo real ("el mejor modelo del mundo es el mundo"). Esta insatisfacción fue mostrada, principalmente, por Rodney Brooks el cual caracterizaba las arquitecturas jerárquicas como una "descomposición horizontal" (Murphy, 2000). En lugar de esto, un estudio etimológico de lo que significaba inteligencia sugirió que dicha inteligencia se estructura de forma vertical.

Figura 2.10: Arquitectura reactiva

Bajo esta descomposición vertical, un agente comienza con primitivos comportamientos de supervivencia y evoluciona a nuevos tipos de comportamiento, desde los más simples hasta los más avanzados. Cada uno de estos comportamientos tiene acceso independiente a los sensores y actuadores, de manera que, si algo sucede con los comportamientos avanzados, los básicos seguirán funcionando de manera correcta (Murphy, 2000). Se dedica entonces el esfuerzo al subsistema sensorial, no a la planificación.

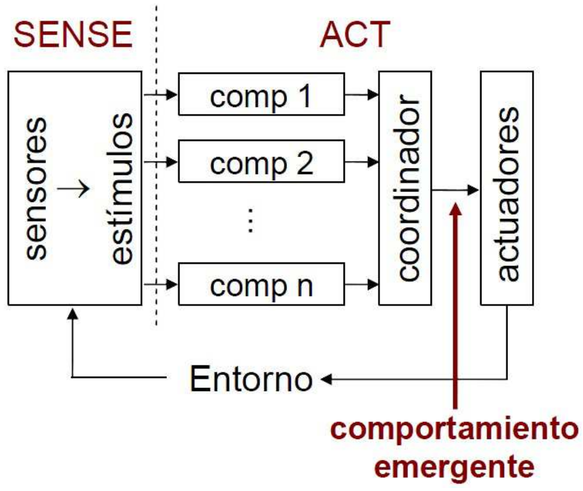


Figura 2.11: Esquema paralelo de una arquitectura reactiva.

De manera concreta, a la hora de realizar un sistema robótico operativo, los comportamientos se realizan en capas que son independientes y que, como se ha comentado, tienen acceso a los sensores y actuadores.

Al principio se dudaba de la eficacia de este paradigma. Los usuarios de robótica no estaban conformes con la imprecisa manera en la que los comportamientos discretos se combinaban con los comportamientos más potentes.
Finalmente, los rápidos tiempos de ejecución asociado a unos comportamientos reflexivos terminaron por convencer a la sociedad robótica y a posteriori surgiría el paradigma híbrido, del que se hablará más adelante.

Asemejando el paradigma reactivo con la estructura jerárquica detección-planificación-actuación, se podría decir que el reactivo omite la parte de planificación, solo utiliza detección-actuación. Las arquitecturas creadas a partir del paradigma reactivo son las encargadas de especificar cómo se coordinan y controlan los comportamientos.

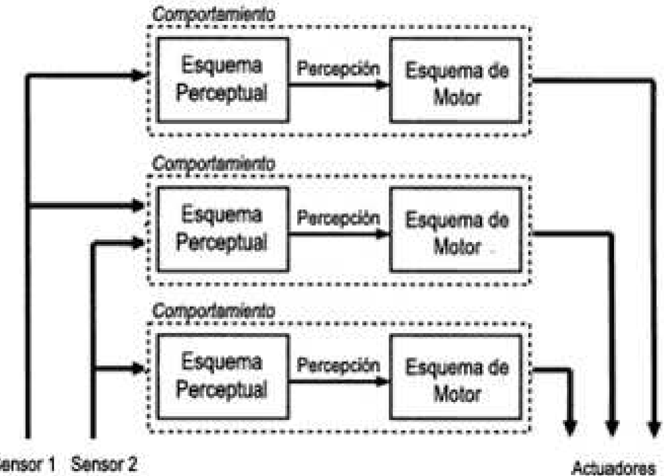


Figura 2.12: Descomposición vertical de tareas (comportamientos) asociada al paradigma reactivo.

Las características principales de este tipo de arquitecturas son:

- La tarea no se descompone en funciones sino en comportamientos.
- No necesita conocimiento a priori del entorno ni modelos del entorno.
- No precisa planificación.
- Son adecuadas para entornos dinámicos y/o parcial o totalmente desconocidos.

Las principales características de los comportamientos reactivos son:

- La salida de cada comportamiento debe ser completamente determinista.
- Cada comportamiento es independiente del resto.
- Los comportamientos no poseen memoria.
- Dan como resultado respuestas innatas a estímulos interiores y exteriores.
- El resultado debe ser homogéneo.

Los principales comportamientos reactivos empleados en algunas arquitecturas son:

- Parada de emergencia.
- Alejarse de objetos cercanos.
- Seguimiento de contornos.
- Evitación de obstáculos.
- Atracción a un punto.
- Ir a zonas libres.
- Movimiento aleatorio.
- Moverse en la misma dirección.

Las principales características de un gestor de comportamientos reactivos son:

- Puede utilizar diferentes mecanismos de gestión: gestión por fusión, por prioridades y mixtos.
- El gestor por prioridades puede estar constituido por un sólo nivel o por varios.
- División de los comportamientos por grupos, gestionándose, en un primer nivel, los comportamientos del mismo grupo. Una posible clasificación es:
    - Comportamientos de seguridad.
    - Comportamientos preventivos.
    - Comportamientos de navegación
    - Comportamientos de desbloqueo
- El gestor por fusión puede utilizar reglas heurísticas, lógica borrosa, redes neuronales,…. Un ejemplo de regla heurística es: si existen obstáculos en la dirección del movimiento, da más peso al comportamiento de evitación, de lo contrario, da más peso al comportamiento de atracción.

Llegados a este punto y, una vez que se ha introducido el paradigma reactivo, es obligatorio tratar sobre la arquitectura que más se ajusta a este paradigma, la arquitectura de subsunción de Rodney Brooks.

Rodney Brooks desarrolló esta arquitectura a mediados de los ochenta en el MIT argumentando que el paradigma jerárquico y sus arquitecturas evolucionaban en detrimento de la construcción de robots de trabajo reales. Decía que la construcción de modelos de mundo y razonamientos usando representaciones simbólicas del conocimiento era un impedimento para el tiempo de respuesta de un robot y llevaba a los investigadores de robótica por el camino equivocado (Arkin, 1999).

Según (Murphy, 2000), la arquitectura de subsunción de Rodney Brooks es la arquitectura basada en el paradigma reactivo más influyente en el mundo de la robótica. En parte, esta influencia fue debida a la publicidad creada de los sistemas generados con esta arquitectura. Los robots creados a partir de esta arquitectura lucen como insectos del tamaño de una caja de zapatos, con seis patas y antenas. Además, estos robots fueron los primeros capaces de caminar, evitar colisiones y trepar sobre los obstáculos sin las pausas típicas “mover-pensar-mover-pensar” de los antiguos robots.

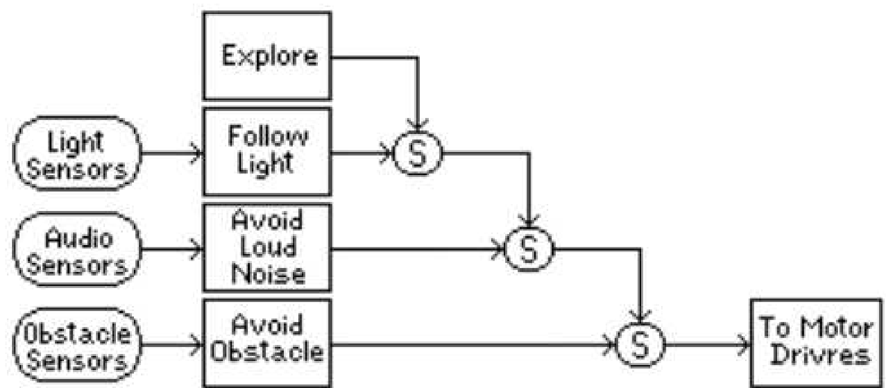


Figura 2.13: Diagrama de bloques de la arquitectura Subsumption

El término comportamiento en la arquitectura de subsunción de Brooks tiene un significado menos preciso que en otras arquitecturas. Un comportamiento es una red de módulos de actuación y detección que realizan una tarea. Estos módulos son máquinas finitas de estados aumentadas las cuales tienen registros, tiempos y otras mejoras que les permiten interactuar con otros módulos (Murphy, 2000). Se puede decir que la máquina de estados finita aumentada es el equivalente a la interfaz entre los esquemas y la estrategia de coordinación de control en un esquema de comportamientos.

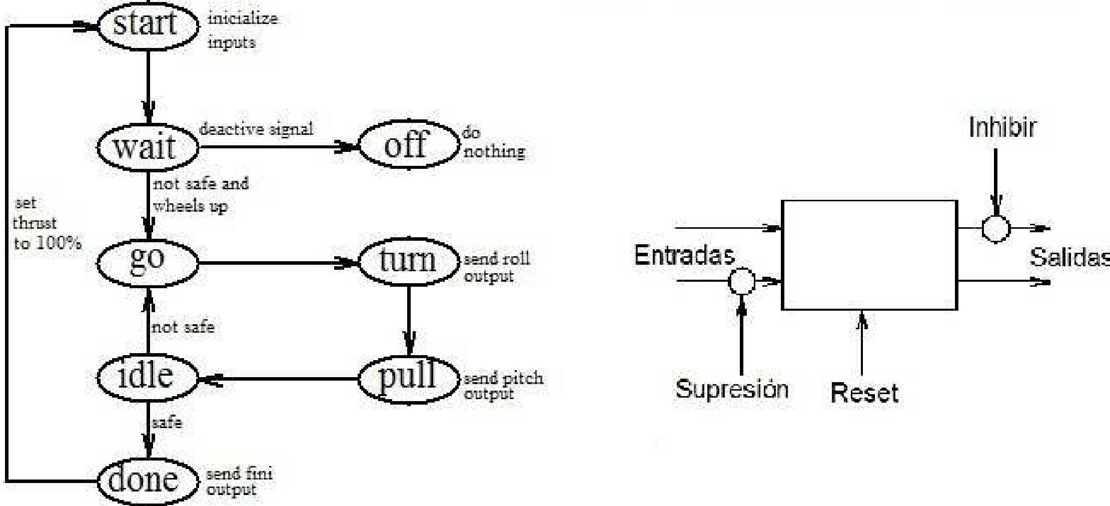


Figura 2.14: Máquina de estado finito en la arquitectura subsunción para la definición de un módulo

Existen cuatro aspectos interesantes cuando hablamos de esta arquitectura:

- Los módulos son agrupados en capas. Las capas reflejan una jerarquía de inteligencia. Las capas de bajo nivel encapsulan funciones de supervivencia básicas mientras que los niveles más altos crean acciones para la consecución de metas. Cada capa puede ser vista como un comportamiento abstracto para una tarea en particular.

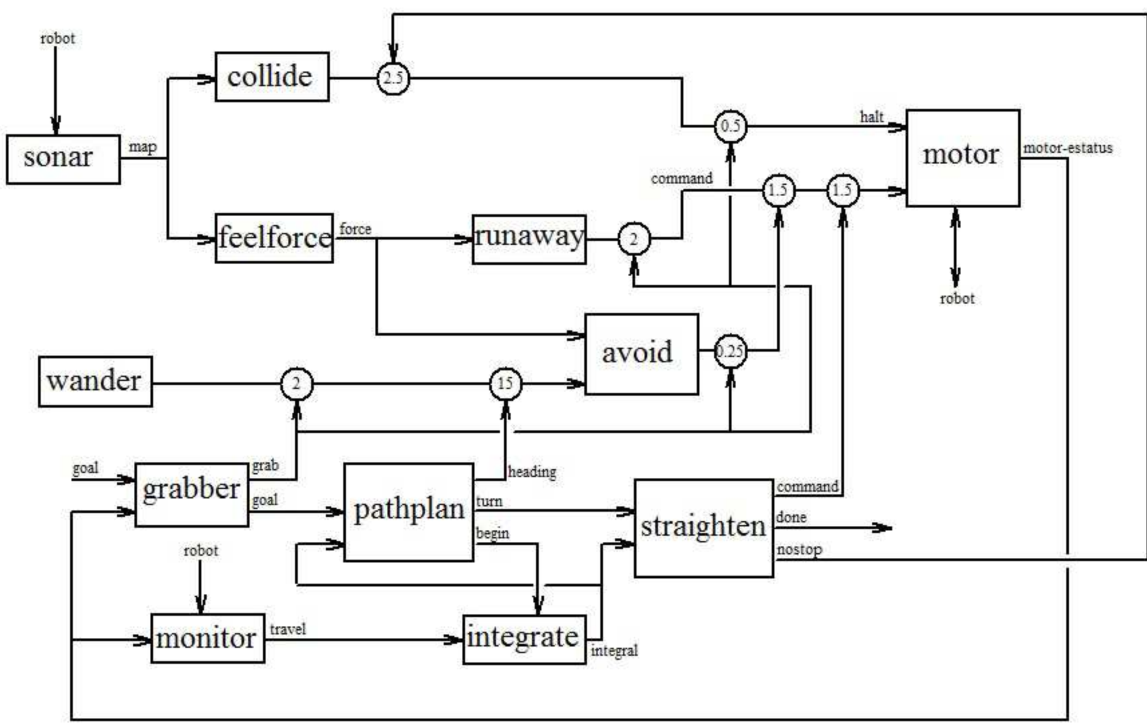


Figura 2.15: Esquema de los tres primeros niveles de la arquitectura

- Los módulos que están en capas altas pueden ser sobrescribir o suprimir la salida por comportamientos que están en la capa inmediatamente anterior. Las capas de comportamiento actúan de manera independiente y concurrentemente por lo que es necesario un mecanismo para manejar los potenciales conflictos. La solución en subsunción es que el ganador es siempre la capa más alta.

- Se evita el uso de estados internos. Se entiende estado interno como cualquier tipo de representación persistente local que muestre el estado del mundo. Como el robot es un agente situado, la gran mayoría de esta información es proporcionada por el mismo entorno en tiempo real.

- Una tarea es completada debido a la activación de la capa que corresponde, la cual activa capas de menor nivel y así continuamente. Sin embargo, los sistemas de subsunción no son fácilmente divisibles en tareas, es decir, no se puede ordenar que se realice otra tarea sin ser reprogramada.

2.3.3. Arquitecturas híbridas

Hacia finales de los ochenta, la tendencia en inteligencia artificial de robots era diseñar y programar basándose en el paradigma reactivo. Pero el coste de aplicar el paradigma reactivo era que se debía eliminar la planificación o cualquier otra función que involucrara memorizar o razonar sobre el estado global del robot en relación a su entorno. Esto significa que un robot no podría planear una ruta optima, crear mapas, monitorizar su propia configuración o seleccionar el comportamiento más adecuado para llevar a cabo la tarea indicada.

Por lo que en los inicios de los años noventa, la comunidad de IA se encontraba en un punto en el que la deliberación y la planificación de rutas habían vuelto al panorama robótico pero, al mismo tiempo, no se debía perder en gran logro que supuso el paradigma reactivo (Murphy, 2000). Esto quiere decir que el paradigma buscado debía estar preparado para actuar correctamente ante entornos variables o desconocidos, con lo que se hace imprescindible contar con un sistema reactivo (Arkin, 1999). Debido a estas razones se hizo necesaria la creación de un paradigma híbrido.

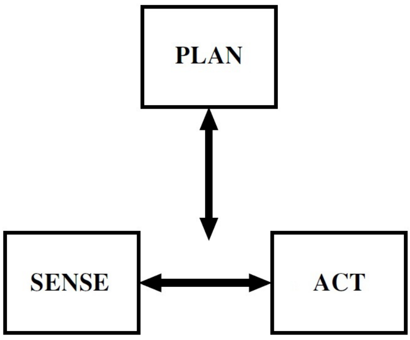


Figura 2.16: Organización del paradigma híbrido

El actual pensamiento de la comunidad robótica es que el paradigma híbrido es la mejor solución general por varias razones. Utiliza técnicas de procesamiento asíncronas permitiendo a las funciones deliberativas ejecutarse independientemente de los comportamientos reactivos (Murphy, 2000).

De hecho, era más que inevitable que, teniendo en cuenta que ambos paradigmas, tanto el reactivo como el deliberativo, contaban con ventajas que cubrían los inconvenientes del otro paradigma, surgiera un paradigma híbrido. La primera fue Mataric, alumna de Brooks, la cual incluyó una capa de planificación en la parte superior del sistema de capas múltiples del paradigma reactivo (Bekey, 2005).

La organización de un sistema híbrido se podría describir como planificar, y después detectar-actuar. El apartado de planificación incluye toda la deliberación y el modelado de un mundo global, no solo tareas y planificación de rutas, es decir, el robot, primero, debe pensar (planificar) como va a conseguir la misión o tarea y después instancia una serie de comportamientos para ejecutar el plan o una parte de él.

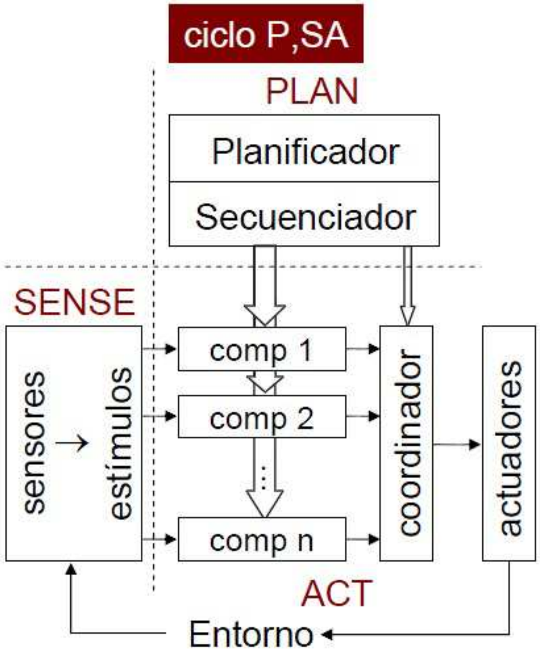


Figura 2.17: Combinación serie y paralelo de módulos en una arquitectura híbrida

La organización de las primitivas detección, actuación y planificación están divididas conceptualmente en una parte reactiva y otra parte deliberativa. Aunque muchas arquitecturas tienen discretas capas de funcionalidades dentro de la parte reactiva y deliberativa, cada arquitectura tiene la parte reactiva y la parte deliberativa. Según (Murphy, 2000) existen tres categorías de arquitecturas híbridas:

- Estilo gestionado: enfocado en la subdivisión de la parte deliberativa en capas basadas en el enfoque de control de cada función deliberativa.
- Jerarquías de estado: usan el conocimiento del estado del robot para distinguir entre comportamientos reactivos y actividades deliberativas. Los comportamientos reactivos se ven sin estado y funciones solo de presente. Las funciones deliberativas pueden ser divididas entre las que necesitan un conocimiento de un estado pasado, anterior y las funciones que requieren conocimiento de un estado futuro.
- Estilos orientados al modelo: Se caracterizan por que los comportamientos tienen acceso a partes del modelo de mundo.

Como ejemplo de este tipo de arquitectura presentamos **AuRA**. Esta arquitectura fue desarrollada por Ronald C. Arkin en el School of Information and Computer Science at the Georgia Institute of Technology, Atlanta ([Arkin88], [Arkin89a], [Arkin89b], [Arkin89c].

Se trata de una arquitectura híbrida en la que existe tanto un control reactivo como un control deliberativo. El control reactivo se basa en la existencia de módulos elementales, denominados ***esquemas***, que implementan comportamientos simples que, mediante combinación, pueden dar lugar a comportamientos complejos. El diagrama general de la arquitectura viene mostrado en la figura 2.18.

Está básicamente definida por cinco subsistemas:

- *Subsistema de Percepción*. Se encarga de obtener y filtrar los datos de los sensores y, estructurarla en una forma que sea útil a los subsistemas de planificación, cartográfico y motor.
- *Subsistema Cartográfico*. Se encarga de gestionar y modificar los mapas temporales que se van creando en función de la información sensorial que se obtiene durante el desarrollo del proceso de control.
- *Subsistema de Planificación*. Está formado por un planificador jerárquico (componentes de planificación) y un gestor de esquemas motor (componentes reactivos).

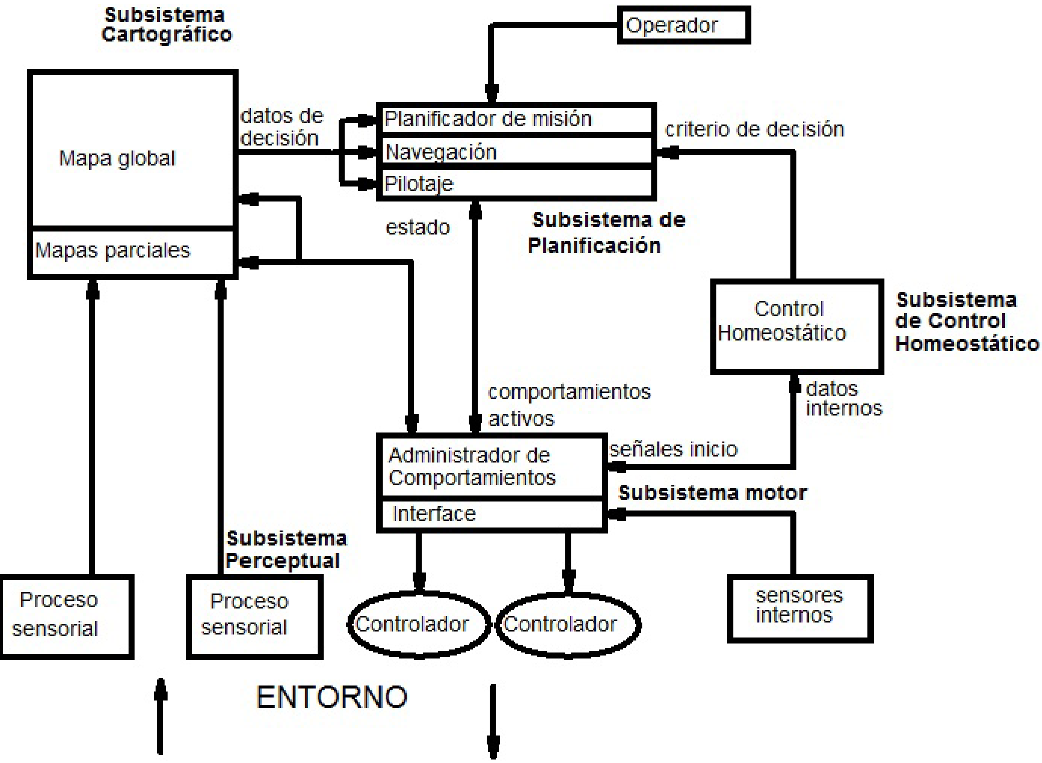


Figura 2.18: Arquitectura Aura

- *Subsistema Motor.* Es la interface software con el robot específico que debe ser controlado.
- *Subsistema Homeostático*. Se encarga de la re-planificar dinámicamente en función de los recursos internos disponibles.

En la figura 2.19 puede verse la relación entre estos esquemas. Podemos observar cómo los esquemas-motor (que implementan determinados comportamientos) poseen esquemas de percepción (que recogen la información que necesitan, suministrada por los sensores ambientales) y esquemas de recepción que reciben información de los sensores internos a través de los esquemas de transmisión.

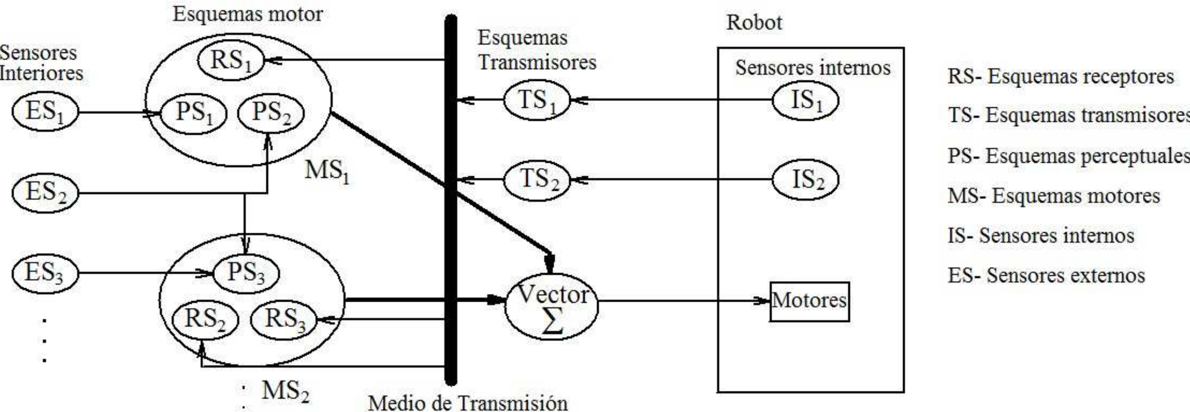


Figura 2.19: Relación entre los diferentes esquemas de la arquitectura Aura

Cada uno de esos esquemas-motor genera un vector velocidad (que se realiza de una forma análoga a la de campos potenciales, en los que el entorno que rodea al robot se divide regularmente en zonas cuadradas o celdas donde sobre cada una de ellas se sitúa un vector que define la fuerza asociada al campo en ese punto del entorno) que son sumados para la obtención de un único vector que defina el comportamiento total del robot en ese momento.

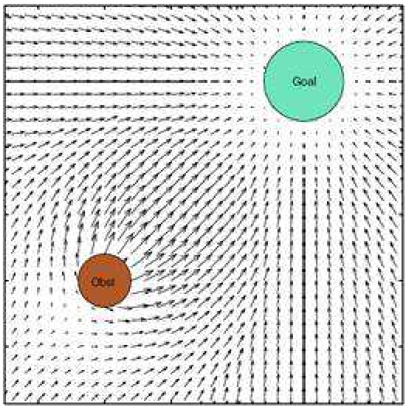


Figura 2.20: Planning utilizando campos potenciales

Como resumen, las características de la arquitectura son:

- Arquitectura hibrida con control reactivo y control deliberativo.
- Red dinámica de comportamientos en vez de capas fuertemente cableadas.
- Basada en estudios psicológicos y neurocientíficos de comportamientos.
- Utiliza un modelo global del entorno para la planificación jerárquica.
- Modificación de comportamientos (esquemas) en función del estado interno del robot.
- Uso de expectaciones y capacidades en el proceso de percepción.
- Uso de campos potenciales en el cálculo de cada comportamiento básico

2.4. Subsistemas y módulos típicos para una arquitectura híbrida para robots móviles

Una vez analizados los fundamentos que rigen cada una de las arquitecturas robóticas, en este apartado se muestra como se organizan los distintos subsistemas y módulos que componen el sistema, para el caso concreto de una arquitectura híbrida.

En (Goodwin, 2008) se puede obtener una clasificación muy coherente y bien estructurada de los componentes más comunes de un robot móvil, categorizados según las áreas que el autor de este documento ha estructurado según se observan en la figura 2.20. El autor ha intentado agrupar las catergorías en tres capas horizontales: reactiva ejecutiva y deliberativa. En la figura 2.22 se profundiza en cada uno de los subsistemas de la figura 2.20:

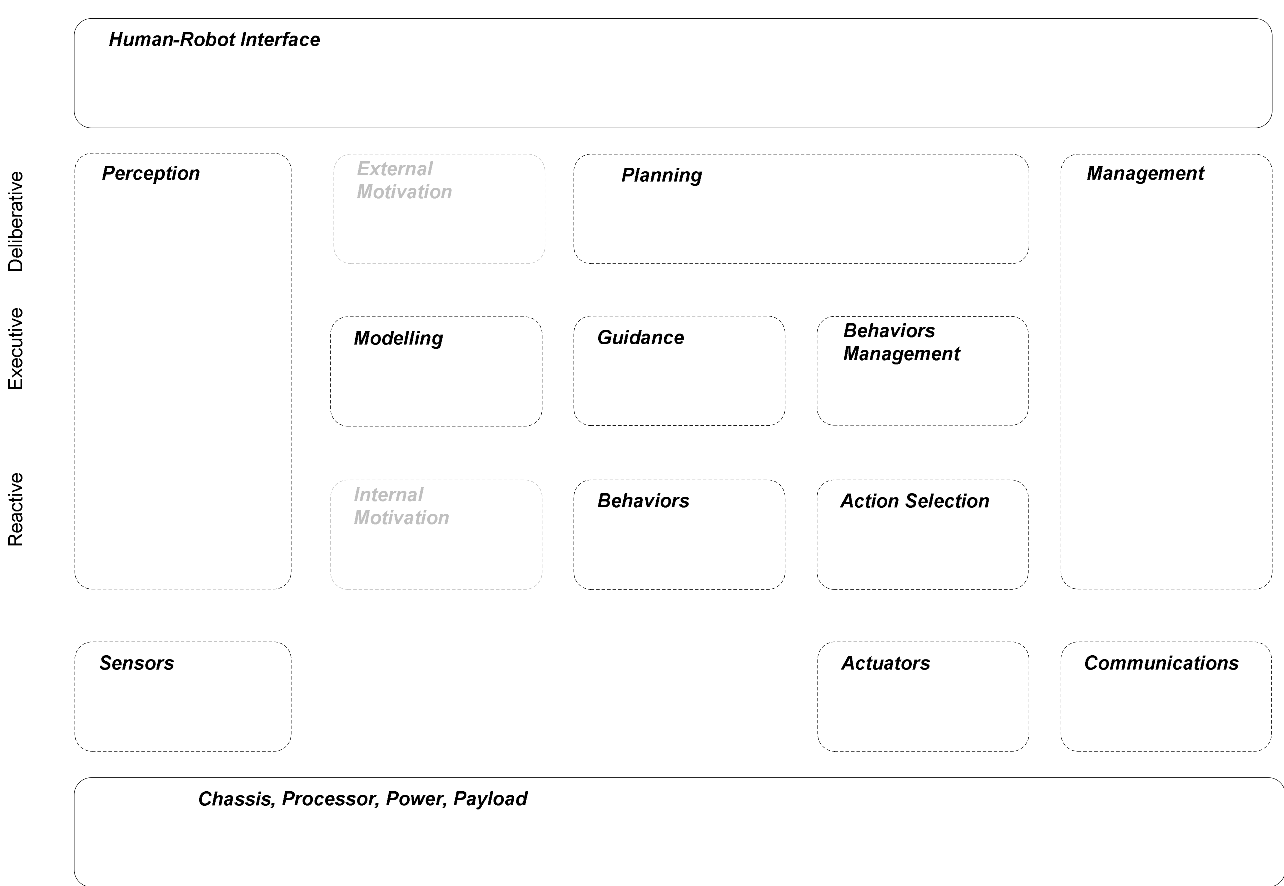


Figura 2.21: Subsistemas típicos en una arquitectura híbrida para robots móviles, distribuidos según tres capas. Los subsistemas "Internal / External Motivation" se representan en gris porque podrían ser parte de "Behaviors" o de "HRI" respectivamente.

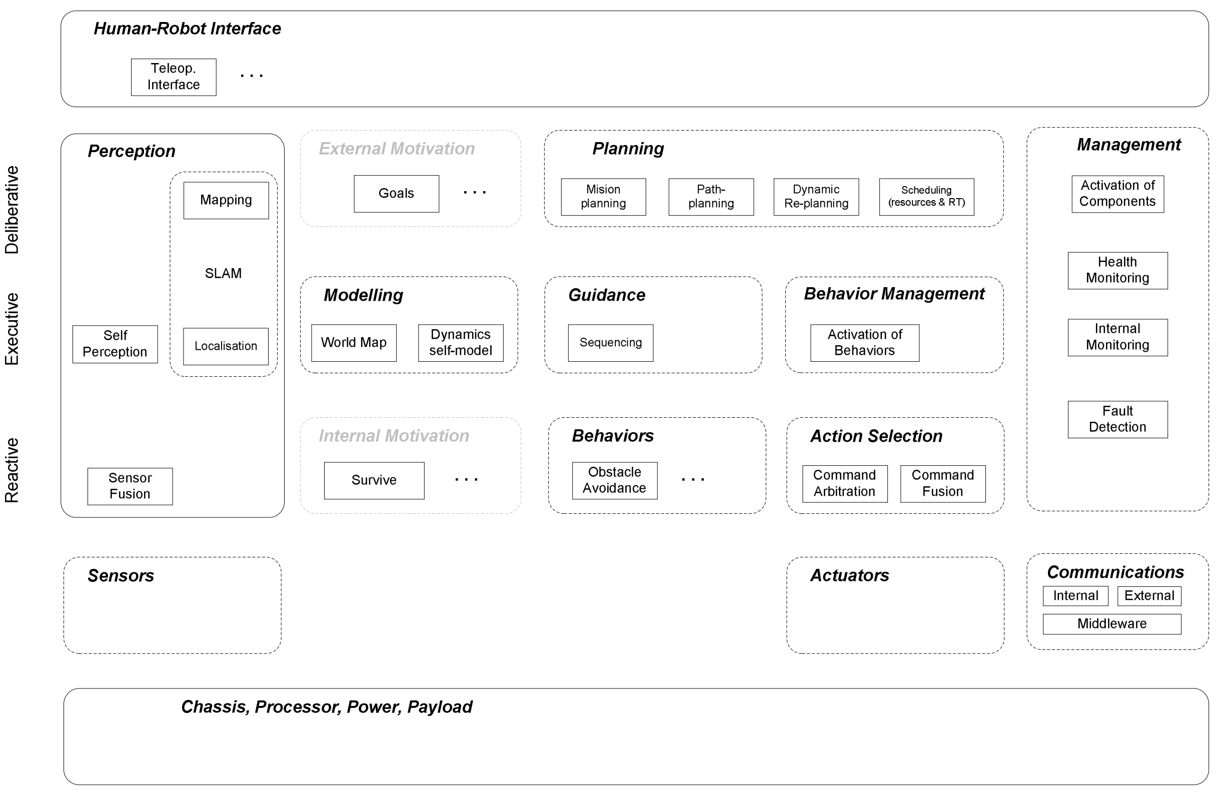


Figura 2.22: Componentes típicos de cada subsistema de una arquitectura híbrida para robots móviles.

A continuación se resume la finalidad de cada uno de los subsistemas:

- ***Human Robot Interaction (HRI)***
  Cubre todos los aspectos de la interacción humana con uno o más robots, incluyendo la operación y desarrollo del robot. Algunos de los dispositivos HRI más comunes son: interfaz de control remota, interfaz de tele-operación, *waypoint interface*, interfaz de configuración, visualización de datos…
- ***Perception***
  La percepción del entorno es proporcionada por los sensores y otros robots. La percepción incluye la construcción de mapas, la auto-percepción, localización y Sensor Fusion.
- ***Modelling***
  El mapa y la localización proporcionada por el subsistema de percepción, ofrecerán una representación del entorno que será utilizado por otros subsistemas.
- ***Planning***
  Los planificadores se pueden utilizar para realizar las siguientes operaciones: *Path planning* (determina una ruta optima entre el comienzo y el último estado), *Sequencing* (secuencia un orden en el que realizar las metas), *Sharing* (compartir objetivos entre robots), *Resource Scheduling* (para gestionar el uso de los recursos)…
- ***Management***
  Utilizado para gestionar la activación y desactivación de los distintos módulos, monitorización interna del sistema, detección de fallos y supervisión del estado.
- ***Behaviors management***
  Utilizado para gestionar la activación y desactivación de los distintos comportamientos y módulos. Se incluye como parte del nivel o capa ejecutiva.
- ***Guidance***
  La orientación es utilizada para determinar la posición actual exacta en un proceso de planificación y proporcionar así la siguiente acción.
- ***Motivation***
  Permite al operador del robot proporcionar un conjunto de objetivos (*External Motivation*). El robot elegirá dependiendo de sus propias motivaciones (*Internal Motivation*), como puede ser la seguridad del vehículo, llevar a cabo todas o algunas de estos objetivos.
- ***Behaviour***
  Proporcionan una o más acciones cuando se les proporciona un estímulo concreto por parte de los sensores del vehículo o módulo *Perception.*
- ***Action Selection***
  Módulo que realiza la toma de decisiones en busca de la acción más apropiada.
- ***Actuators***
  Encargados de realizar los movimientos del vehículo o manipuladores cuando se les ordena una acción.
- ***Communications***
  Encargado de las comunicaciones entre módulos distribuidos, robots y dispositivos del HRI.
- ***Chasis***
  Proporciona la estructura hardware del robot.
- ***Processor***
  Lleva a cabo todas las tareas de procesamiento.
- ***Power***
  Proporciona energía a todos los componentes del robot. Esto puede incluir a los sistemas de la recolección, transferencia y almacenamiento.

- ***Payload***
  Permite al robot transportar objetos o herramientas, tales como otros robots.

Una vez se han definido los distintos módulos que forman parte del sistema robótico, el siguiente objetivo será realizar la implementación de cada uno de ellos. En este apartado se describen las distintas posibilidades que tiene el usuario para poder llevar a cabo estas implementaciones.

2.5. Aproximaciones reutilizables para programar aplicaciones robóticas

Para afrontar la implementación de los distintos módulos de un sistema robótico haremos uso de un paradigma de programación determinado. Existe una infinidad de definiciones de lo que es un paradigma. Una posible es que un paradigma es un determinado marco desde el cual miramos el mundo, lo comprendemos, lo interpretamos e intervenimos sobre él. Es la manera como percibimos el mundo.

Un paradigma de programación provee (y determina) la visión y métodos de un programador en la construcción de un programa o subprograma. Diferentes paradigmas resultan en diferentes estilos de programación y en diferentes formas de pensar la solución de problemas (con la solución de múltiples "problemas" se construye una aplicación). Algunos de los paradigmas de programación más habituales son:

- Imperativo o por procedimientos: Describe la programación como una secuencia instrucciones o comandos que cambian el estado de un programa. Ejemplos de lenguajes imperativos son *C, BASIC y Pascal.*
- Funcional: Concibe la computación como la evaluación de funciones matemáticas y evita declarar y cambiar datos. En otras palabras, hace hincapié en la aplicación de las funciones y composición entre ellas, más que en los cambios de estados y la ejecución secuencial de comandos. Algunos ejemplos de lenguajes funcionales son *Haskell y Scheme.*
- Lógico: El paradigma lógico se basa en la definición de reglas lógicas para luego, a través de un motor de inferencias lógicas, responder preguntas planteadas al sistema y así resolver los problemas. El lenguaje de programación lógica por excelencia es *Prolog.*
- Orientado a objetos: Paradigma que utiliza objetos (entidades que tienen un determinado estado, métodos e identidad) y sus interacciones, para diseñar aplicaciones y programas informáticos. Algunos ejemplos de lenguajes orientados a objetos son *Smalltalk, Java y C++.*
- Orientado a componentes: Paradigma que descompone el sistema en componentes funcionales lógicos con interfaces bien definidas usadas para la comunicación entre componentes.

Los lenguajes de programación están basados en uno o más paradigmas de programación con el objetivo de que un programador utilice el más conveniente a la hora de resolver un problema. Ningún paradigma es capaz de resolver todos los problemas de forma sencilla y eficiente, por lo tanto es útil poder elegir entre distintos "estilos" de programación dependiendo del tipo de problema.

También hay lenguajes que permiten mezclar los paradigmas que, en principio, parecerían irreconciliables. Se debe aclarar que hay subparadigmas que se incluyen en paradigmas más generales, pero hay otros que utilizan métodos de programación totalmente distintos entre sí e igualmente hay lenguajes que los combinan. Por ejemplo, el lenguaje Oz emplea programación lógica, funcional, orientada a objetos y otras. Lenguajes como Delphi, C++ y Visual Basic combinan el paradigma imperativo, el procedural y el orientado a objetos.

Para implementar las arquitecturas descritas en los apartados anteriores *(2.3 – Clasificación arquitecturas robóticas)* en un robot se pueden adoptar las distintas aproximaciones o paradigmas de programación mencionados. Si bien es cierto algunos de estos paradigmas actualmente están en completo desuso, y otros no son recomendables para el desarrollo de software en el campo de la robótica.

Probablemente el paradigma de programación actualmente más utilizado a todos los niveles es el orientado a objetos**. La programación orientada a objetos o POO** (OOP según sus siglas en inglés), supuso durante la década pasada, un cambio radical en el proceso de desarrollo de software.

La POO difiere de la programación estructurada tradicional, en la que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos de entrada para obtener otros de salida.

La programación estructurada anima al programador a pensar sobre todo en términos de procedimientos o funciones, y en segundo lugar en las estructuras de datos que esos procedimientos manejan. En la programación estructurada sólo se escriben funciones que procesan datos. Los programadores que emplean POO, en cambio, primero definen objetos para luego enviarles mensajes solicitándoles que realicen sus métodos por sí mismos.
Los mecanismos de encapsulación de POO soportan un alto grado de reutilización de código, que se incrementa por sus mecanismos de herencia.

De un proceso caracterizado por su énfasis en la descripción algorítmica de la solución del problema, se pasó a un proceso orientado a la representación y manipulación de los objetos que caracterizan el problema.

La **creación de programas para robots** debe cumplir ciertos requisitos específicos frente a la programación en otros entornos más tradicionales. En los últimos años los avances en aplicaciones de robots móviles autónomos han sido significativos. Sin embargo, durante mucho tiempo se ha considerado que la integración en el desarrollo de software para la requiere únicamente un esfuerzo menor, una vez que los algoritmos necesarios están disponibles. Con bastante frecuencia las dificultades a superar han sido enormemente subestimadas.

Esto acaba produciendo que algoritmos que son conocidos y siempre utilizados en las distintas aplicaciones, son implementados una y otra vez desde cero dando lugar a la pérdida de valiosos recursos. Por tanto, en el desarrollo de software para aplicaciones robóticas los desarrolladores siempre buscarán con gran ahínco la reutilización completa de sus diseños. La reutilización de código se refiere al comportamiento y a las técnicas que garantizan que una parte o la totalidad de un programa informático existente se pueda emplear en la construcción de otro programa. De esta forma se aprovecha el trabajo anterior, se economiza tiempo, y se reduce la redundancia.

En los siguientes apartados analizamos los distintos enfoques actuales, que buscan alcanzar un alto grado de reutilización. Estos enfoques no son, en general, excluyentes entre sí, sino que una misma aplicación puede incluir varios de ellos (por ejemplo en los frameworks orientados a objetos que utilizan un middleware, algunos de ellos incluyen el diseño del middleware dentro del propio framework, mientras que otros por el contrario utilizan un middleware diseñado por terceros).

2.5.1. Frameworks OO

Este paradigma abrió nuevas posibilidades para desarrollar software basado en la noción de reutilización de módulos. La Orientación a Objetos creó un rumbo diferente en el proceso de reutilización a través de la producción de módulos genéricos, fáciles de integrar, distribuidos e independientes de las plataformas de desarrollo.

En el desarrollo de software robótico existen librerías específicas para este campo, que son utilizadas por las subrutinas y también realizan un intento de maximizar en lo posible la reutilización del software. Una librería es un conjunto de recursos (estructuras de datos, funciones, etc.) que el usuario puede usar en sus programas. El efecto de incluir una librería es básicamente extender el conjunto de recursos que provee el lenguaje, sin ninguna lógica o jerarquía subyacente, sencillamente agrego componentes nuevos a este. Sin embargo no es suficiente con reutilizar librerías.

La **utilización de frameworks aporta un paso más hacia la reutilización de las estructuras en las que se organiza el software**.

> Un framework define, en términos generales, un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular, que sirve como referencia para enfrentar y resolver nuevos problemas de índole similar (reutilización de código).

Típicamente, puede incluir soporte de programas, librerías, y un lenguaje interpretado, entre otras herramientas, para así ayudar a desarrollar y unir los diferentes componentes de un proyecto. Representa una arquitectura de software que modela las relaciones generales de las entidades del dominio, y provee una estructura y una especial metodología de trabajo, la cual extiende o utiliza las aplicaciones del dominio. Las principales ventajas que ofrecen los framework son la reducción del coste de los procesos de desarrollo de aplicaciones software para dominios específicos, y la mejora de la calidad del producto final (Fayad y Schmidt, 1997).

Según Gamma, el framework determina la arquitectura de la aplicación [Gamma,1995]. Este es un buen enfoque, ya que el framework se encarga de definir la estructura general, sus particiones en clases y objetos, las responsabilidades clave, así como la colaboración entre dichas clases y objetos. Todos estos parámetros son definidos por el framework, evitando que el usuario tenga que definirlos y se pueda centrar en cosas específicas de su aplicación. "El framework captura las decisiones de diseño que son comunes a su dominio de aplicación" (Gamma, 1995). De esta manera, un framework no solo promueve la reutilización de código sino también la reutilización de diseño.

Una librería y un framework están concebidas para distintos usuarios. La primera puede ser usada por cualquiera que necesite el tipo de objetos o algoritmos que ésta presta como recursos, mientras que la segunda debe usarse para desarrollar una aplicación concreta del dominio. Es útil pensar en una librería como un gran paquete de piezas (recursos), con las que el usuario arma un rompecabezas determinado (su aplicación), existiendo muchísimas formas distintas de hacerlo.

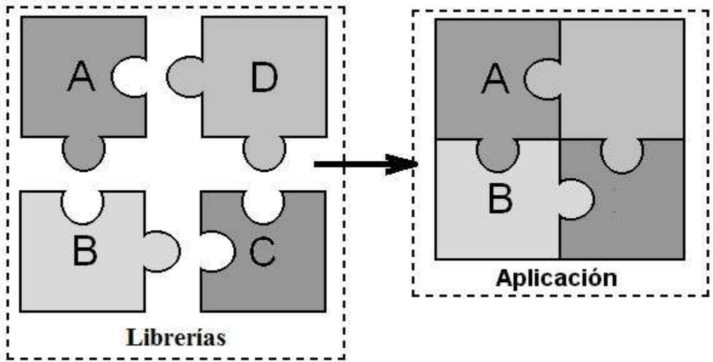


Figura 2.23: Utilización de una librería

Un framework tiene una concepción radicalmente distinta. Primero identifica las partes variables de una aplicación a otra del dominio (*Hot Spots*), y tras definir su arquitectura e interfaces, arma el resto del rompecabezas de modo que el usuario solo tenga que preocuparse de la implementación de esos huecos o *Hot Spot* donde se define realmente el comportamiento que tendrá la aplicación.

El equivalente gráfico expone de forma clara la diferencia entre uno y otro. Mientras que con el uso de librerías el programador define todo el diseño de la aplicación y la construye ayudándose de estas librerías, en un framework esta estructura está básicamente establecida y el usuario solo es responsable de codificar las porciones distribuidas a través de la arquitectura que varían de una aplicación a otra.

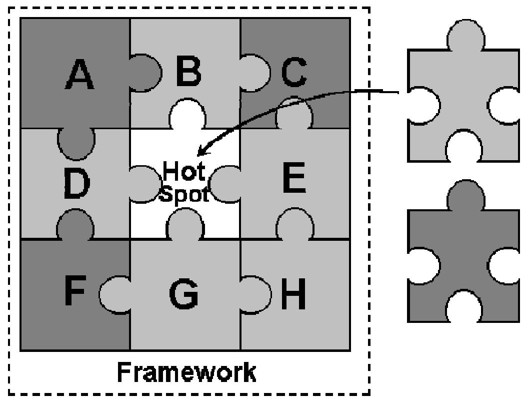


Figura 2.24: Utilización de un framework

Destacamos que usar un framework es usar su arquitectura, así que esta se re-usa de una aplicación a la siguiente. Un framework es adecuado para re-usar código si este queda integrado al mismo. También hay que destacar que el uso de un framework requiere un conocimiento más detallado de la arquitectura, o sea, el usuario de un framework no puede pensar en sentarse a programar sin haber leído previamente y en detalle, la documentación del sistema. La utilización de un framework es una buena inversión para una persona que desarrolle software en un área específica, ya que el diseño tiene resueltos la mayoría de los problemas de esta.

Una vez definidos los frameworks así como las ventajas que pueden proporcionar para construir aplicaciones, podemos dividirlos en horizontales y verticales según sea el tipo de aplicación que acometan.

Los **frameworks verticales** son aquellos desarrollados específicamente para un dominio de aplicación concreto, y cubren un amplio espectro de aplicaciones, desde las telecomunicaciones (TINA [TINA-C, 1995]), la fabricación [Schmidt, 1995], servicios telemáticos avanzados (MultiTEL [Fuentes y Troya, 1999])... Por ser muy específicos y requerir un conocimiento muy preciso de los dominios de aplicación resultan los más costosos de desarrollar.

Los **frameworks horizontales** están dedicados a modelar infraestructuras de comunicaciones [Schmidt, 1997a][Hüni, 1997], interfaces de usuario [Taylor, 1996], los entornos visuales [Florijn, 1997] y, en general, cualquiera de los aspectos relacionados con el sistema subyacente [Bruegge, 1993].

Como ejemplo de framework orientado a objetos de propósito general dedicado a modelar infraestructuras de comunicaciones podemos presentar ACE. **Adaptative Communication Environment (ACE)**, es un framework orientado a objetos de código abierto, que implementa muchos de los patrones centrales para el desarrollo de software de comunicación. ACE es independiente de la plataforma, existiendo implementaciones en Windows, Linux, Mac, etc.

ACE está implementado en C++ y se ha orientado desde sus principios a desarrolladores de aplicaciones y servicios de alto rendimiento en red y tiempo real. ACE separa el sistema operativo de la aplicación a través de la capa de Adaptación al Sistema Operativo.

Dicho de otra manera, ACE es una implementación de sockets independiente de la plataforma final de ejecución (los sockets se utilizan de forma distinta en máquinas Linux y Windows). ACE proporciona una implementación distinta para cada plataforma, aunque el uso de ACE será siempre igual (al igual que ocurre con la JVM). En la figura 2.25, aparece el diagrama que ilustra los componentes clave en ACE y sus relaciones jerárquicas.

ACE fue inicialmente desarrollado por Douglas C. Schmidt durante su trabajo de posgrado en la Universidad de California. Hoy en día, la mayoría de los trabajos de desarrollo de ACE se realizan en el Instituto de Software de Sistemas Integrados (ISIS) de la Universidad Vanderbilt. http://www.cs.wustl.edu/~schmidt/ACE.html

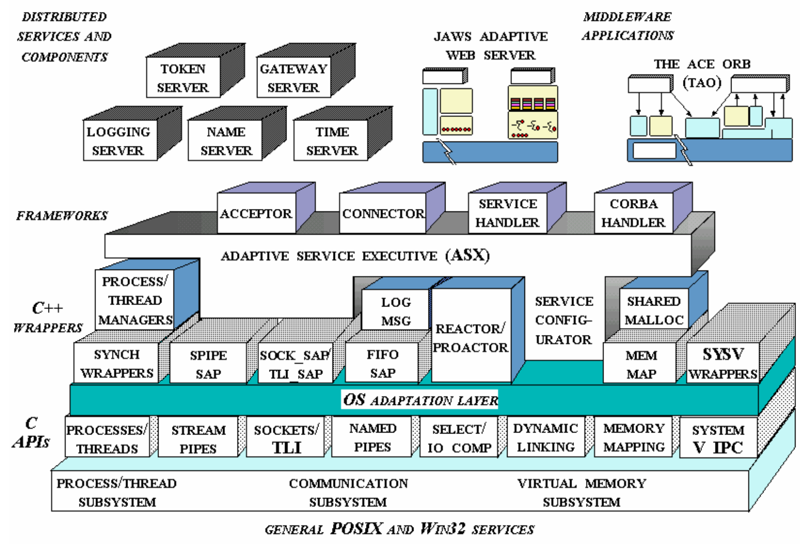


Figura 2.25: ACE y sus componentes

Fruto de la necesidad de que los frameworks integren las comunicaciones, aparecen los denominados Marcos de Trabajo Distribuidos (***Middleware Application Frameworks****),* diseñados para integrar módulos y aplicaciones software en ambientes distribuidos, permitiendo altos niveles de modularidad y reutilización en el desarrollo de nuevas aplicaciones, y aislando la mayor parte de las dificultades conceptuales y técnicas que conlleva la construcción de aplicaciones distribuidas [Fayad y Schmidt, 1997].

Un middleware permite el desarrollo de software modular mediante la implementación de un protocolo estándar de comunicaciones que permite el intercambio de información entre los distintos módulos. Estos procesos pueden estar ejecutándose en el mismo procesador o en un procesador diferente.

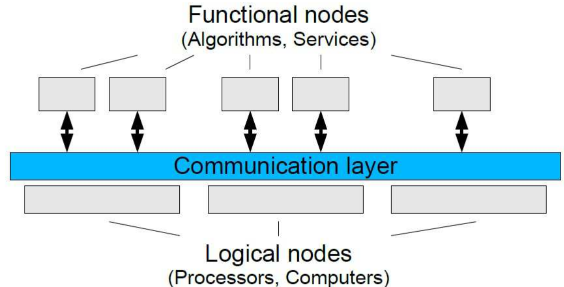


Figura 2.26: Capa de comunicación entre componentes (middleware)

Alcanzado este punto resulta interesante hablar de **CORBA**. CORBA (Common *Object Request Broker Architecture* — arquitectura común de intermediarios en peticiones a objetos), es un estándar que establece una plataforma de desarrollo de sistemas distribuidos facilitando la invocación de métodos remotos. Se trata de una infraestructura de computación de objetos distribuidos normalizada por el Object Management Group (OMG) http://www.omg.org/

CORBA define las APIs, el protocolo de comunicaciones y los mecanismos necesarios para permitir la interoperabilidad entre diferentes aplicaciones escritas en diferentes lenguajes y ejecutadas en diferentes plataformas, lo que es fundamental en computación distribuida.

En un sentido general, CORBA "envuelve" el código escrito en otro lenguaje, en un paquete que contiene información adicional sobre las capacidades del código que contiene y sobre cómo llamar a sus métodos. Además CORBA utiliza un lenguaje de definición de interfaces (IDL) para especificar las interfaces de los servicios que los componentes ofrecerán. La figura 2.30 ilustra los componentes principales de la arquitectura de modelo de referencia OMG.

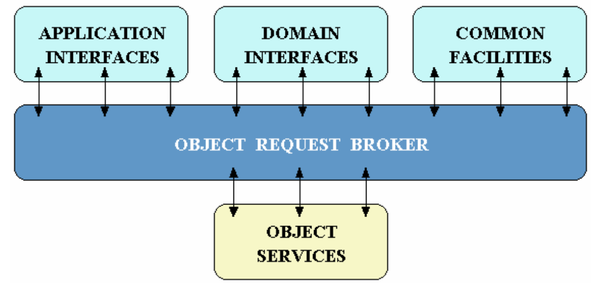


Figura 2.27: OMG modelo de referencia de la arquitectura

Object Request Broker (también por sus siglas ORB) es la capa de software que permite a los objetos realizar llamadas a métodos situados en máquinas remotas, a través de una red. Maneja la transferencia de estructuras de datos, de manera que sean compatibles entre los dos objetos.

Para ello utiliza un estándar con el que convertir las estructuras de datos en un flujo de bytes, conservando el orden de los bytes entre distintas arquitecturas. Este proceso se denomina marshalling (y también su opuesto, unmarshalling). Básicamente permite a objetos distribuidos interactuar entre sí de manera transparente, es decir, como si estuviesen en la misma máquina. La figura 2.31 ilustra los componentes principales de la arquitectura CORBA ORB.

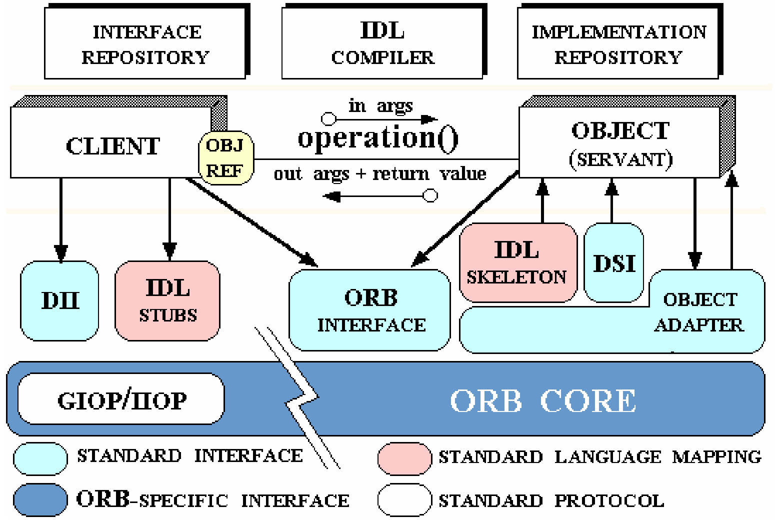


Figura 2.28: Arquitectura CORBA ORB

Sin embargo CORBA no es más que un estándar, aunque existen diversas implementaciones del mismo. A continuación se describen dos de ellas.

TAO (***The ACE ORB***) es una implementación de CORBA utilizando ACE para el acceso a sockets. TAO es de libre distribución, de código abierto y sigue fielmente los estándares para las implementaciones de la especificación de Real-Time CORBA, con lo que provee de una QoS (*Quality of Service*) *extremo a extremo* eficiente, previsible y fiable.

TAO permite a los clientes invocar operaciones sobre objetos distribuidos sin preocuparse por la localización de dichos objetos, lenguaje de programación, plataforma de sistema operativo, protocolos de comunicación e interconexiones, e independiente de plataforma.

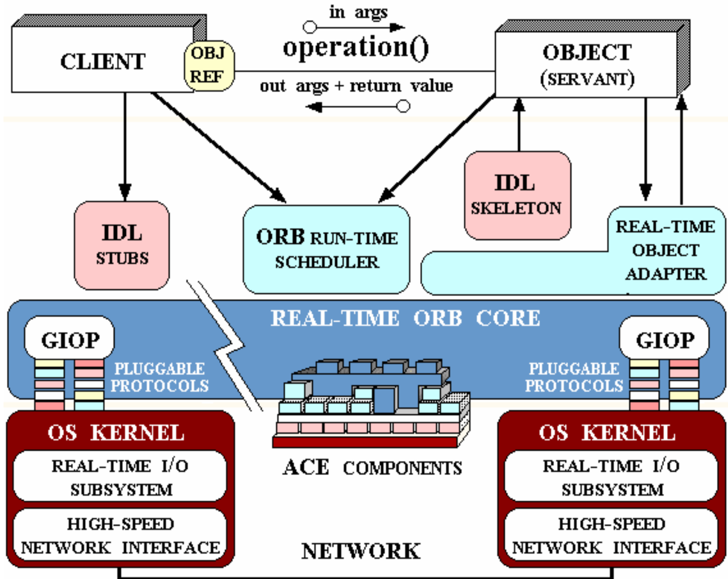


Figura 2.29: The ACE ORB (TAO)

Otro ejemplo es **ICE** (*Internet Communications Engine*). ICE es un middleware orientado a objetos que puede ser utilizado para aplicaciones de Internet sin la necesidad de utilizar el protocolo HTTP, siendo capaz de atravesar cortafuegos a diferencia de otros middlewares. El diseño de ICE está claramente influenciado por CORBA (de hecho fue creado por varios desarrolladores CORBA). Sin embargo, es más pequeño y menos complejo que CORBA.

ICE es un conjunto de CORBA, que incluye los componentes para la invocación remota de objetos, la replicación, la computación grid, failover, balanceo de carga, firewall recorridos y servicios de publicación-suscripción. Para acceder a esos servicios, las aplicaciones están relacionadas con una biblioteca de código auxiliar o el montaje, que se genera a partir de un lenguaje independiente IDL.

Ha sido desarrollado por ZeroC (www.zeroc.com/ice.html) y bajo licencia doble (GNU GPL y licencia propietaria). Es compatible con C++, Java, C#, Python, PHP, etc. en la mayoría de los principales sistemas operativos tales como Linux, Solaris, Windows y Mac OS X.

Frameworks robóticos y middleware

Una vez vistos descritos los frameworks orientados a objetos así como los middlewares, pasamos a presentar algunos ejemplos concretos de frameworks orientados a objetos y middlewares diseñados específicamente para el campo de la robótica. Un estudio sobre el estado de la ingeniería software en la robótica, centrándose en las decisiones de diseño que hay detrás de algunos frameworks se da en la siguiente dirección:
http://wiki.robot-standards.org/index.php/Current_Middleware_Approaches_and_Paradigms

Player - http://playerstage.sourceforge.net/

Player es un servidor de red para el control de robots. La parte principal de este servidor es el protocolo de comunicación personalizado que permite el modelo de comunicación cliente-servidor. Player puede ser visto como un servidor multiproceso de aplicaciones, que proporciona aplicaciones y servicios a los programas cliente.

Player ofrece una interfaz limpia y sencilla de los sensores del robot y actuadores sobre la red IP. Los programas clientes se comunican a Player través de un socket TCP, leyendo los datos de los sensores,

escribiendo comandos en los actuadores, y la configuración de los dispositivos sobre la marcha; cada uno con su propio hilo se ejecución.

El programa cliente puede ejecutarse en cualquier máquina que tenga una conexión de red al robot, y puede ser escrito en cualquier lenguaje que soporte sockets TCP. Como se puede ver en la figura 2.30, los hilos se comunican a través de un espacio de direcciones global compartido.

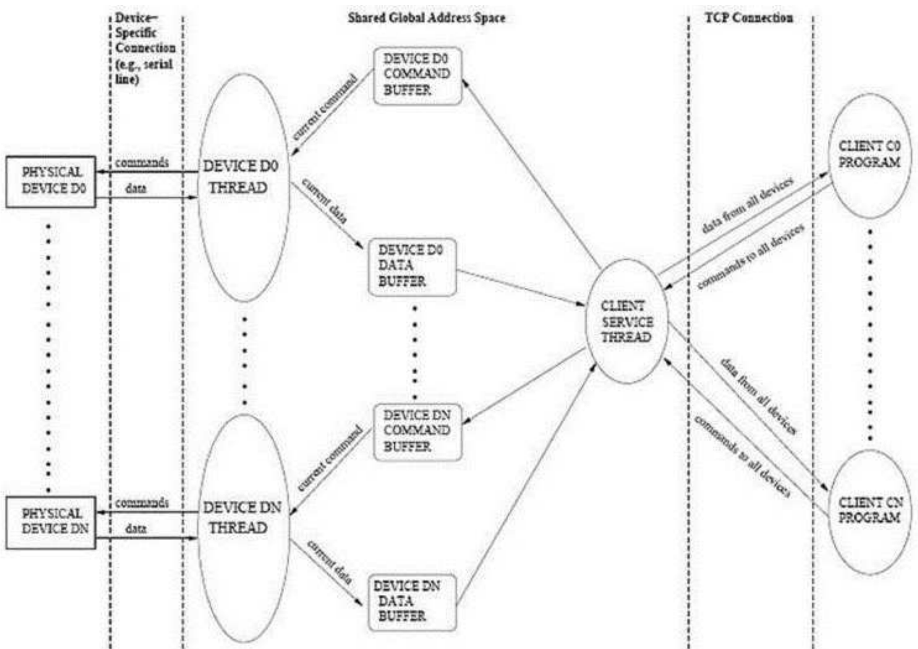


Figura 2.30: Servidor de dispositivos internos de Player

Player es compatible con diversas plataformas de hardware robóticas. La plataforma original es la familia Pioneer2 de *ActiveMedia*, pero actualmente son compatibles otros robots y sensores. La arquitectura modular de Player hace que sea fácil añadir soporte para nuevo hardware, dando lugar a que comunidades de desarrolladores contribuyan al desarrollo de nuevos drivers

Player está diseñado para ser independiente de la plataforma y lenguaje, siendo de código abierto y publicado bajo licencia pública GNU.

MIRO - http://smart.informatik.uni-ulm.de/MIRO/index.html

MIRO es un framework orientado a objetos distribuidos para el control de robots móviles, basado en CORBA. Los módulos básicos de MIRO se han desarrollado en C + + para Linux. Sin embargo, debido a la independencia del lenguaje de programación que aporta CORBA, es posible desarrollarlos en cualquier lenguaje y sobre cualquier plataforma que proporcione una implementación a la especificación. Los componentes básicos de MIRO han sido desarrollados bajo la ayuda de ACE, y utilizan TAO para la comunicación entre ellos.

MIRO divide el sistema en varias capas distintas, como se muestra en la figura 2.31. Las capas superiores sólo pueden acceder a las capas inferiores a través de sus interfaces. En el caso de MIRO, estas capas son las siguientes:

1. *MIRO device layer* - Esta capa ofrece clases para interfaces al hardware abstrayendo de los detalles del hardware de bajo nivel. Estas clases permiten el acceso a los recursos del hardware a través de llamadas simples a procedimientos.

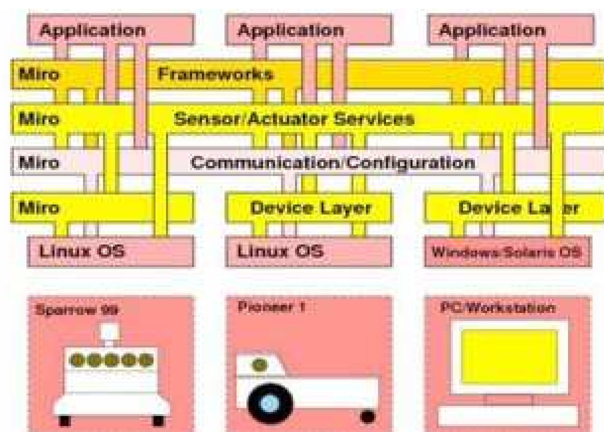


Figura 2.31: Capas de abstracción de MIRO

2. MIRO service layer - Esta capa proporciona a los servicios abstracción de los sensores y actuadores a través del lenguaje de definición de interfaces de CORBA (IDL). Estos servicios se implementan como objetos CORBA de red. Las clases de este nivel presentan los sensores y actuadores como servicios genéricos.

3. MIRO framework layer - En esta capa se proporcionan módulos funcionales específicos de robótica. Ejemplos de ello son la cartografía, localización y planificación de ruta.

El principio utilizado para crear jerarquías complejas de comportamientos es similar a una máquina de estados finitos representada en la figura. Los denominados “Patrones de acción”, se componen de comportamientos y "guardias" que pueden notificar acerca de algún evento externo. Los autores afirman que una reconfiguración dinámica de las políticas es posible y que esta característica también puede contribuir a la solidez.

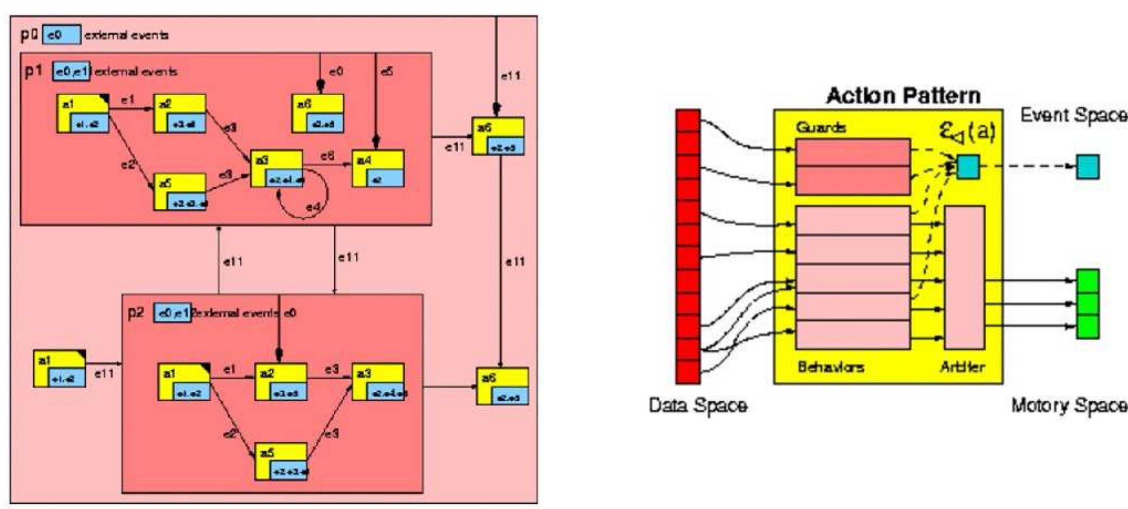


Figura 2.32: Patrones de acción organizados en una jerarquía de comportamientos

Webots API - http://www.cyberbotics.com/

Webots es un entorno de desarrollo utilizado para modelar, programar y simular los robots móviles. El sistema está organizado como una interfaz de programación orientada a objetos para el control de robot. Los objetos corresponden a los dispositivos de robot: ruedas diferenciales, cámara, sensor de distancia, range-finder, acelerómetro, servo, etc Para cada objeto, se definen una serie de métodos. En la figura 2.33 se muestran un subgrupo de la API de Webots.

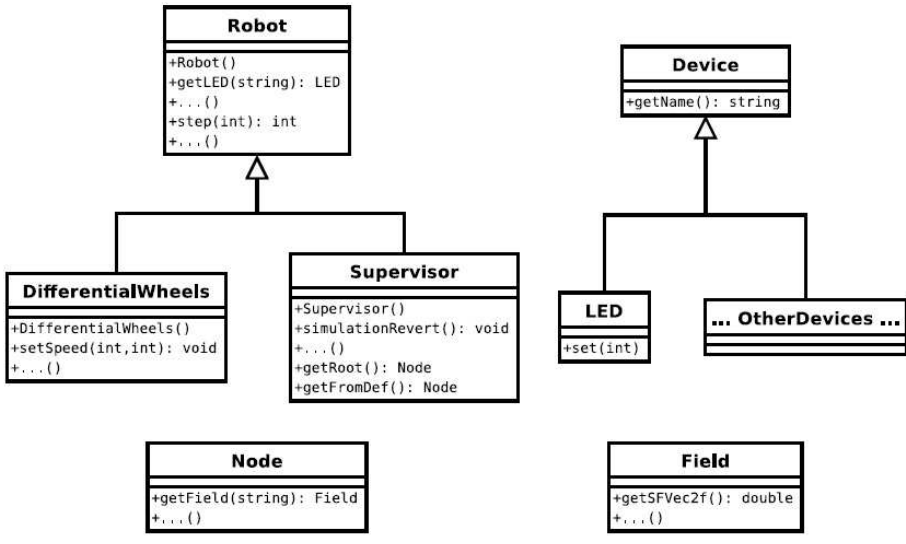


Figura 2.33: Pequeño subgrupo de la API orientada a objetos de Webots

Existen por ejemplo, métodos para activar y desactivar las mediciones del sensor, sensor de recuperación de información, establecer los parámetros, el envío de comandos a un motor, etc

Esta API orientada a objetos está disponible en varios lenguajes de programación, incluyendo C, C++, Java y Python. También hay una interfaz TCP / IP (escrito en C) que permite programar un robot utilizando la API de Webots desde cualquier programa de terceros compatible con TCP / IP (es decir, Matlab con la caja de herramientas TCP / IP, Labview, Lisp, etc. Ver figura 2.34).

El protocolo de comunicación entre la API y el robot utiliza tres tecnologías diferentes dependiendo si se utiliza la API Webots para la simulación, para el control remoto o un programa transversal compilado:

La API Webots soporta un gran número de dispositivos y robots. Aún así la API está totalmente documentada, por lo que es posible ampliarla y adaptarla para el control de nuevos robots. Las especificaciones y todas las implementaciones de la robótica API Webots son software libre de código abierto.

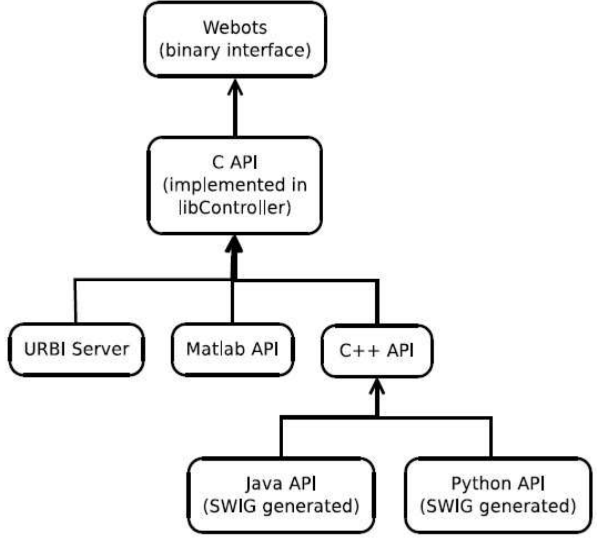


Figura 2.34: API Webots

2.5.2. Frameworks OC

La Programación Orientada a Objetos (POO) ha sido el sustento de la ingeniería del software durante muchos años. Sin embargo, se ha mostrado insuficiente al tratar de aplicar sus técnicas para el desarrollo de aplicaciones en entornos extensibles. En particular:

1. No permite expresar claramente la distinción entre los aspectos computacionales y meramente composicionales de la aplicación.
2. Hace prevalecer la visión de objeto sobre la de módulo, estos últimos como unidades de composición independientes de las aplicaciones.
3. No tiene en cuenta los factores de mercadotecnia necesarios en un mundo real, como la distribución, adquisición e incorporación de esos módulos a los sistemas.

A partir de estas ideas y elevando el nivel de abstracción, aparece la programación orientada a componentes (POC) como una extensión natural de la orientación a objetos para los entornos extensibles [Nierstrasz, 1995][Szyperski y Pfister, 1997]. La POC nace con el objetivo de construir un mercado global de componentes software, cuyos usuarios son los propios desarrolladores de aplicaciones que necesitan reutilizar componentes ya hechos y probados para construir sus aplicaciones de forma más rápida y robusta.

Este paradigma propugna el desarrollo y utilización de componentes reutilizables dentro de lo que sería un mercado global de software. Sin embargo, disponer de componentes no es suficiente tampoco, a menos que seamos capaces de reutilizarlos. Y reutilizar un componente no significa usarlo más de una vez, sino que implica la capacidad del componente de ser utilizado en contextos distintos a aquellos para los que fue diseñado.

En este sentido, uno de los sueños que siempre ha tenido la ingeniería del software es el de contar con un mercado global componentes, al igual que ocurre con otras ingenierías como es el caso de los circuitos integrados (*Integrated Circuit*), donde los componentes (IC) se puedan comprar e integrar fácilmente en nuevos diseños siempre que se conozcan claramente sus interfaces y patrones de interacción con los mismos.

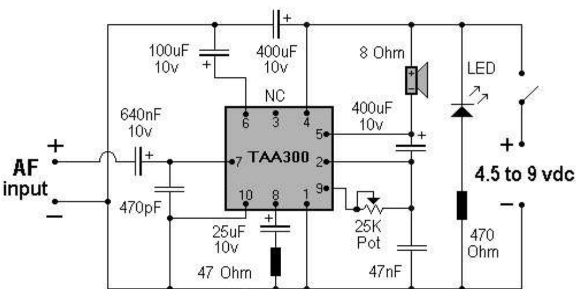


Figura 2.35: Reutilización de componentes electrónicos en el diseño de un dispositivo

Las entidades básicas de la POC son los componentes, "cajas negras" que encapsulan cierta funcionalidad y que son diseñadas para formar parte de ese mercado global de componentes, sin saber quién los utilizará, ni cómo, ni cuándo.

Los usuarios conocen acerca de los servicios que ofrecen los componentes a través de sus interfaces y requisitos, pero normalmente ni quieren ni pueden modificar su implementación. Una ***definición de componente*** por (Szyperski, 1998):

"Un componente es una unidad de composición de aplicaciones software, que posee un conjunto de interfaces especificadas contractualmente y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio".

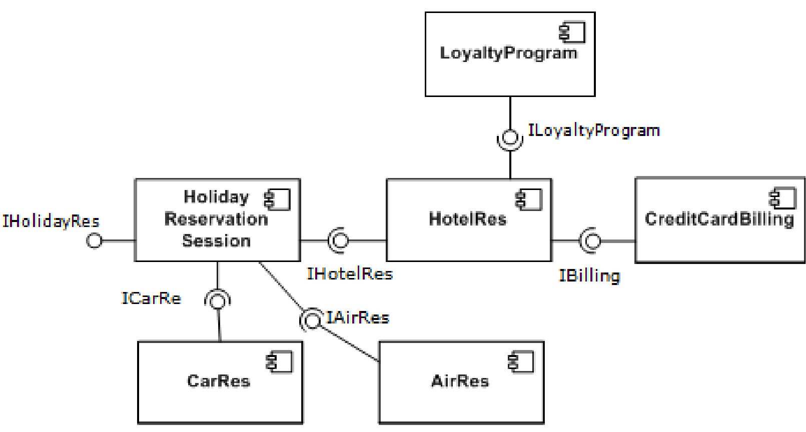


Figura 2.36: Hipotético sistema de reserva de vacaciones representado mediante componentes e interfaces en UML

Una vez disponemos del concepto de componente, un ***modelo de componentes*** define la forma de sus interfaces y los mecanismos para interconectarlos entre ellos. La coordinación e interacción entre componentes exige un marco regulatorio estandarizado (modelo de componentes) que establece la infraestructura de software requerida (framework) y las convenciones y restricciones de diseño de los mismos.

Los principales modelos de componentes existentes los constituyen COM+/Microsoft, Enterprise JavaBeans/Sun y el modelo de componentes CORBA (Heineman & Councill, 2001). El primero es propietario y no portable entre sistemas operativos, el segundo sólo está disponible en Java y el último aún carece de implementaciones fiables y maduras. Esta opinión también es compartida por (Brooks, 2005).

A menudo, los modelos de componentes son bastante complejos y requieren importantes conocimientos de programación, ya que no ofrecen ninguna ayuda o consejos sobre la mejor forma de definir las interfaces de los componentes visible desde el exterior. Esto a menudo es de dominio específico y su completa comprensión requiere de cierta experiencia.

Es exactamente este tipo de conocimiento el que tiene que ser puesto a disposición de todo aquel interesado, de tal manera que las mejores prácticas puedan ser fácilmente adoptadas por los recién llegados. Más tarde, ellos podrán contribuir a la robótica de una manera coherente, sin dejarse desanimar por el obstáculo de la complejidad del software.

Una ***interfaz*** define el conjunto de operaciones que un componente puede realizar; estas operaciones son llamadas también servicios o responsabilidades. Las interfaces proveen un mecanismo para interconectar componentes y controlar las dependencias entre ellos. El que los componentes poseen interfaces requeridas es el rasgo principal que los distingue de los objetos.

Con el fin de definir y fijar el aspecto de las interfaces que tendrán los componentes, nace el concepto de patrones de comunicación. Hacer uso de los patrones de comunicación como medio para poder dominar la diversidad de las interfaces de los componentes, se propuso por primera vez en (Schlegel y Wörz, 1999). Continuando con este enfoque y tras haber madurado a lo largo de varios proyectos, encontramos una descripción más completa en (Schlegel, 2004). Fruto de todo este trabajo nacerá SmartSoft.

Dentro de la POC se está trabajando en varios frentes: la adecuación de los lenguajes de programación y modelos de objetos existentes para que incorporen estos conceptos; el diseño de nuevos lenguajes y modelos de componentes; la construcción de herramientas de desarrollo y marcos de trabajo para componentes… En cuanto a los lenguajes de programación, sólo hay unos pocos que realmente incorporen conceptos suficientes para realizar una programación orientada a componentes: *Oberon, Java, Ada95, Modula-3 y Component Pascal*. Sin embargo, ninguno de ellos incorpora todos los conceptos, sino solamente unos pocos.

Tras definir la programación orientada a componentes así como los modelos de componentes, podemos describir los requisitos a tener en cuenta para construir un framework robótico orientado a componentes. De igual manera es posible distinguir los distintos roles de los usuarios que harán uso del framework.

Roles y requisitos de un framework orientado a componentes

Existen diferentes categorías de usuarios, y se distinguen en que cada uno realiza un enfoque diferente sobre la gestión de la integración en la robótica:

Los **usuarios finales** utilizan una aplicación basada en la interfaz de usuario que se les ha proporcionado. Se centran en la funcionalidad de su aplicación y hacen uso de un sistema para cumplir las tareas requeridas. Ellos no se preocupan sobre cómo se ha construido la aplicación por el constructor de las aplicaciones, esperando en todo momento un funcionamiento fiable.

Los **constructores de aplicaciones** realizan (montan) aplicaciones basadas en componentes apropiados y reutilizables. Se encargan de adaptar las aplicaciones mediante el ajuste de parámetros y, a veces incluso completar en la aplicación partes dependientes llamados puntos calientes.
Ellos esperan que el marco (framework) garantice interfaces de componentes claramente estructuradas y coherentes para facilitar el montaje de los componentes testeados fuera de la plataforma (off-the-shelf).

Los **constructores de componentes** se centran en la especificación e implementación de un solo componente. Ellos esperan que el marco (framework) proporcione la infraestructura necesaria con la que se consiga el aprovechamiento de la implementación de un componente de tal manera que sea compatible con otros, sin estar demasiado limitados con respecto a otros componentes internos. Los constructores de componentes quieren centrarse en los algoritmos y funcionalidad de los componentes sin preocuparse de los problemas de integración.

Los **constructores del framework** diseñan e implementan el marco de trabajo, de tal manera que coincida con los múltiples requisitos de la mejor forma posible, permitiendo que cada uno de los anteriores tipos de usuarios pueda centrarse en el papel que le corresponde.

En el ámbito de la robótica, se presentan importantes características que son requeridas y que van más allá del software basado en componentes estándar. Algunos de los requisitos funcionales y no funcionales son:

El uso de un patrón de **cableado dinámico** permite la configuración dinámica de conexiones entre los servicios de los componentes en tiempo de ejecución. Hacer tanto el flujo de control, como el flujo de datos configurables desde el exterior del componente, es la clave para la composición de habilidades, y es necesario en casi cualquier arquitectura robótica. El patrón del cableado dinámico interactúa estrechamente con las primitivas de comunicación y produce una de las principales diferencias con otros enfoques.

El **asincronismo** es un concepto poderoso para separar las actividades y para reducir las latencias mediante la explotación de la concurrencia en la medida de lo posible. El desacoplo es especialmente importante a nivel de componentes para evitar las dependencias de tiempo en la transmisión entre los componentes. Un framework robótico debe aprovecharse del asincronismo siempre que sea posible sin involucrar al usuario del framework.

Las **interfaces de los componentes** tienen que estar definidas con un razonable nivel de granularidad. Con esto conseguiremos evitar una interacción entre componentes fine-grained o de grado fino a la vez que dar soporte a componentes débilmente acoplados; pero con una semántica de interfaz estricta y estándar

Las **estructuras internas de los componentes** pueden seguir diseños totalmente diferentes y los constructores de componentes por tanto, pedir el menor número de restricciones posible. Un framework por tanto tiene que permitir diferentes arquitecturas internas para los componentes. No obstante, ha de garantizar la interoperabilidad, ayudando en la estructuración e implementación de un componente.

Un framework tiene que proporcionar un cierto nivel de transparencia al ocultar los detalles con lo conseguir reducir la complejidad. De la misma manera, hacer totalmente ocultos todos los aspectos de distribución no es deseable ya que a menudo no sólo se traduce en una disminución del rendimiento sino que también impide el no poder pronosticar del tiempo necesario para la comunicación y del uso de los recursos del sistema.

Un **sencillo uso** permite centrarse en la robótica, y consigue que se pueda hacer uso de la reciente tecnología de software que está disponible sin la necesidad de que un experto en robótica deba convertirse en un experto en ingeniería de software. Un ejemplo de los temas a los que se tendrá que hacer frente es por ejemplo la ubicación transparente de los componentes y sus servicios, y los conceptos de concurrencia incluyendo la sincronización y la seguridad de los hilos.

Temas difíciles que tienen que ser abordados son, por ejemplo, la transparencia de localización de componentes y de sus servicios, o conceptos de la concurrencia como la sincronización, exclusión mutua y la seguridad de los subprocesos sin ignorar los diferentes requisitos de ancho de banda (un ejemplo son los sistemas de visión). Un framework proporciona un valor añadido solamente si su uso es mucho más simple que el reconocimiento de todos los requisitos sin necesidad de utilizar un marco.

Un framework será aceptado por los expertos en robótica si ofrece una obvia plusvalía, requiere una curva de aprendizaje plana, se trata de una plataforma software y no de una arquitectura robótica, aborda problemas de middleware y sincronización; y proporciona gran cantidad de controladores de dispositivos y componentes de los sensores y plataformas más frecuentemente utilizados.

Se han propuesto muchos y muy diferentes frameworks software de robótica hasta ahora. La mayoría de ellos poseen un enfoque orientado a objetos sin un modelo de componente explícito (Montemerlo, 2003, Vaughan, 2003, Utz, 2002), otros imponen una estructura de componente interna en particular (Mallet, 2002) o incluso determinan una arquitectura específica de la robótica (Konolige, 1997). Ninguno de ellos ayuda a los constructores de componentes en la definición de mecanismos adecuados para la interacción entre componentes.

Un estudio sobre el estado de la ingeniería software en la robótica centrándose en las decisiones de diseño que hay detrás de algunos frameworks orientados a componentes se da en la siguiente dirección:(http://wiki.robot-standards.org/index.php). A continuación se describen algunos frameworks orientados a componentes.

OROCOS - http://www.orocos.org

Orocos (*Open Robot Control Software)* es un RSF (Robotic Software Framework) impulsado a principios de la década con el objetivo de unificar la metodología y fomentar la estandarización en el desarrollo de arquitecturas software para sistemas robóticos.

Está organizado en forma de cuatro librerías C++:

1. El kit de herramientas en tiempo real (RTT) - Proporciona la infraestructura y funcionalidad para construir componentes en tiempo real.
2. Librería de cinemática y dinámica (KDL) - Proporciona primitivas para el cálculo de cadenas cinemáticas.
3. Librería de filtrado Bayesiano (BFL) - Proporciona un framework para llevar a cabo la inferencia utilizando redes bayesianas dinámicas.
4. Librería de componentes OROCOS (OCL) - Ofrece algunos componentes listos para utilizar.

Los componentes de OROCOS están construidos con RTT, pero también es posible utilizar las funcionalidades proporcionadas por otras bibliotecas. Los usuarios pueden interactuar con el componente a través de su conjunto predefinido de interfaces. Como se puede ver en la figura 2.36, existen cinco maneras de interactuar con un componente.

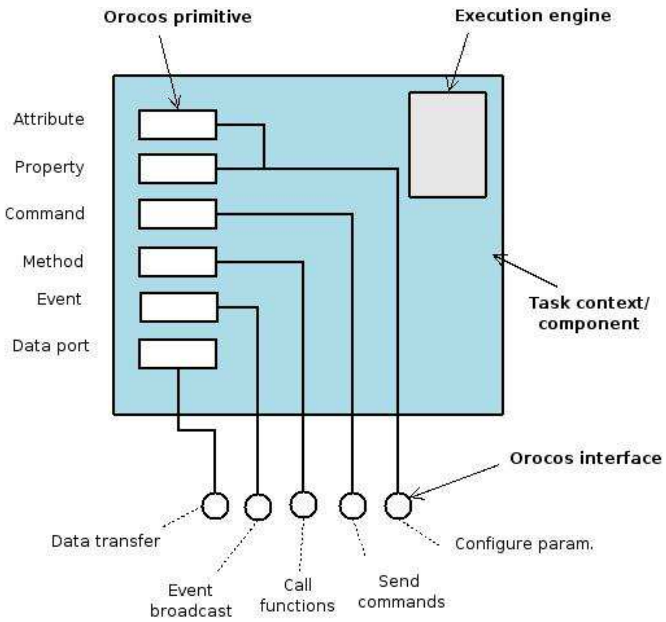


Figura 2.37: Componente OROCOS

Además de definir la interfaz y mecanismos de comunicación de los componentes, OROCOS permite al constructor de aplicaciones implementar máquinas de estado jerárquicas utilizando estos mecanismos. Las máquinas de estados se pueden cargar en tiempo de ejecución en cualquiera de los componentes.

Todos los componentes OROCOS se implementan utilizando la clase "TaskContext", la cual define un "contexto" en el que ejecutar la tarea de los componentes. Los elementos más importantes de un componente OROCOS son:

1. Comandos - Son enviados de manera asíncrona desde un componente a otros componentes (receptores), para instruirlos y lograr un objetivo particular.
2. Métodos - Llamadas realizadas a unos componentes particulares para llevar a cabo algunos cálculos.
3. Propiedades - Parámetros modificables de un componente en tiempo de ejecución que se almacenan en formato XML.
4. Eventos - Son ejecutados por un componente una vez que se produce un cambio particular.
5. Puertos de datos – Mecanismo de comunicación para comunicar buffers de datos entre los componentes.

Para aplicaciones distribuidas OROCOS utiliza TAO (implementación de CORBA) como mecanismo de transporte. Cada tarea definirá sus puertos de intercambio con un nombre único, el tipo de datos que transmite y el método de transmisión.

Orca2 - http://orca-robotics.sourceforge.net/

Orca es un framework de código abierto para el desarrollo de sistemas robóticos basado en componentes. Cada componente es un proceso independiente en tiempo de ejecución. Orca no impone ninguna arquitectura en los componentes ni al sistema en su conjunto.

Todas la comunicaciones entre componentes en Orca2 es manejado por el "Internet Communication Engine de ZeroC. ICE es un middleware de comunicación de uso general, que requiere que todas las interfaces sean definidas explícitamente en un lenguaje de definición de interfaz llamado Slice.

Estas definiciones sirven para describir que servicios ofrecidos y requeridos por los componentes. Internamente, ICE puede ser configurado para usar TCP, UDP, o SSL, como mecanismos de transporte. En la figura 2.37 aparece un ejemplo de un sistema en Orca donde se utiliza el simulador Player/Stage.

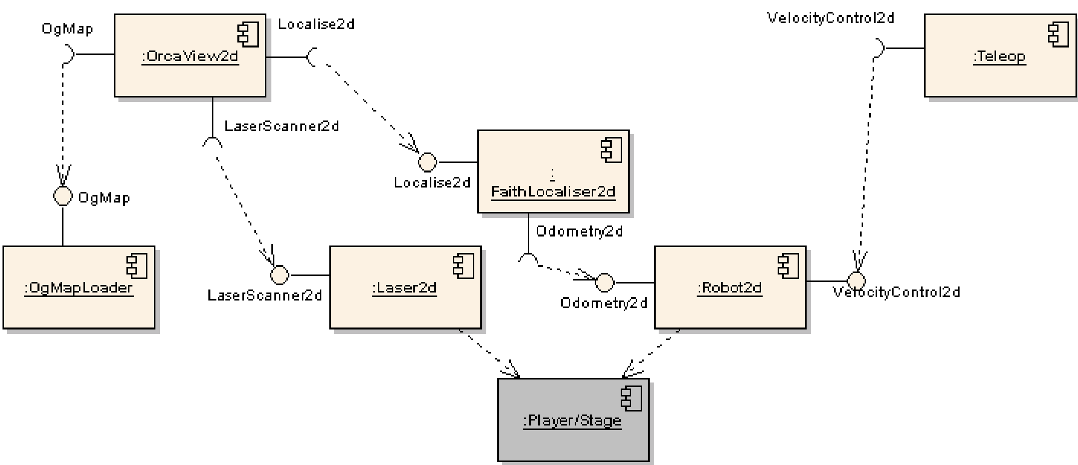


Figura 2.38: Ejemplo de sistema en Orca

Versiones anteriores de Orca soportaban varios mecanismos de transporte (CORBA Ace/Tao y dos bibliotecas de comunicaciones personalizadas). Según el equipo de desarrollo de Orca, el paso a la utilización de un solo middleware comercial de propósito general fue motivado por las dificultades de usar CORBA y las limitaciones que presentaban las comunicaciones personalizadas.

El despliegue de Orca se realiza a través del sistema de construcción CMake[1], y del middleware de comunicaciones ICE. Entre los posibles sistemas operativos con los que podemos trabajar encontramos: Linux, MacOS X, y algunas versiones de Windows. Por otro lado los lenguajes de programación soportados son: C++, Java, C #, Python, etc.

Marie - *http://marie.sourceforge.net/wiki/index.php/Main_Page*

Marie es una herramienta de desarrollo software orientado a componentes para crear sistemas robóticos mediante la integración de componentes software ya existentes y nuevos, desarrollada por la Universidad de Sherbrooke de Canadá. Marie busca conseguir un alto grado de reutilización integrando distintas librerías/paquetes como pueden ser: Player/Stage/Gazebo, FlowDesigner/ RobotFlow, CARMEN/PMAP, ACE, MPICH2, ARIA/ARNL, libxml2, etc.

Marie logra la abstracción adoptando el enfoque de la máquina abstracta que aparece en la figura 2.38. Los autores identifican los siguientes tres niveles:

1. Nivel de aplicación - Se compone de herramientas de integración para crear aplicaciones utilizando los componentes disponibles.
2. Nivel de componentes - Se compone de un framework(s) utilizados para implementar los componentes.
3. Nivel del núcleo - Se compone de utilidades de bajo nivel para las comunicaciones, manejo de datos, dispositivos E/S y cuestiones relacionadas con el sistema operativo.

Desde la perspectiva de la arquitectura Marie utiliza el denominado "Mediator Design Pattern" (MDP) como concepto base de diseño. MDP crea una unidad de control centralizado (mediador) que interactúa con cada componente de forma independiente. El mediador coordina cómo estas aplicaciones interactúan unos con otros.

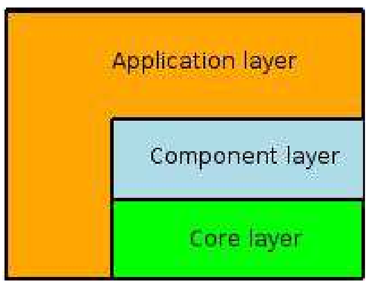


Figura 2.39: Niveles de abstracción de Marie

Para implementar los elementos de diseño los desarrolladores Marie identifican cuatro elementos principales (la unidad Mediator también es referida como Mediator Interoperability Layer(MIL)). En la figura 2.39 aparece como se distribuyen estos elementos.

1. Application adapter (AA) - Responsable del envío y recepción de las solicitudes de servicio y comunicación "MIL→aplicaciones" y "aplicaciones→MIL".
2. Communication adapter (CA) – Responsabe de convertir los datos entre diferentes métodos de comunicación: Mailbox (envíos asíncronos), SharedMap (envía los datos de entrada a varias salidas.), Splitter (estructuras push –in/pull-out) y Switch (solo una entrada al puerto de salida).
3. Communication Manager (CM) - Responsable de crear y administrar los vínculos de comunicación entre los miembros de AA que necesitan estar conectados, es decir, que se utilizan para conectar CA. Debe haber un CM para el nodo de procesamiento en un sistema distribuido.
4. Gestor de aplicaciones (AM) - Responsable del control y la gestión de todo el sistema, mediante la coordinación de los estados del sistema. También es el encargado de configurar y controlar todos los componentes disponibles en el sistema. Debe haber un AM por nodo de procesamiento en el sistema distribuido.

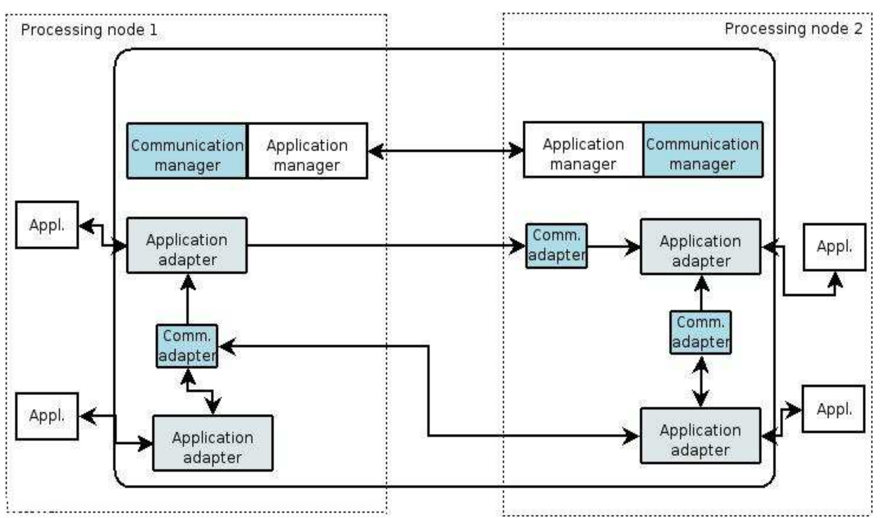
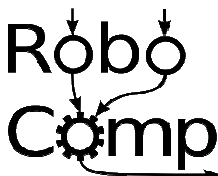


Figura 2.40: Unidad de control centralizada de Marie

Desde el punto de vista del apoyo de middleware, Marie no se vincula a ningún mecanismo de comunicación específico, sino más bien trata de soportar al mayor número posible. Esto se logra por la abstracción del framework, *"port"*, que proporciona los protocolos de comunicación e interconexiones entre componentes. Marie utiliza ACE para implementar la comunicación, por lo que son posibles distintos tipos comunicación (en su mayoría basados en el protocolo TCP mediante sockets).

RoboComp - http://robocomp.sourceforge.net/



RoboComp es un framework robótico desarrollado en la Universidad de Extremadura, de código abierto que ofrece diferentes componentes para el desarrollo de software robótico. Los componentes que se ejecutan forman un gráfico de procesos que pueden ser distribuidos en diferentes núcleos o CPU's utilizando la tecnología de los componentes.

RoboComp no es sólo un conjunto de componentes software, sino que también proporciona un conjunto de herramientas que facilita la creación de nuevos componentes. *RoboComp* no impone ninguna restricción de arquitectura de los componentes o del sistema en su conjunto.

La comunicación entre componentes en *RoboComp* es manejado por *Internet Communication Engine* (ICE) de *ZeroC*, siguiendo el enfoque adoptado por el sistema robótico *Orca2*. *ICE* es un middleware de comunicación de uso general que requiere que todas las interfaces sean definidas explícitamente en un lenguaje de definición de interfaz llamado Slice, y que puede ser configurado para usar TCP, UDP, o SSL, como mecanismos de transporte.

RoboComp utiliza en algunos componentes, herramientas o bibliotecas como *CMake, Qt4, IPP, OpenGL y OpenSceneGraph*. *RoboComp* se puede implementar en cualquier sistema informático que soporte *ICE* (el middleware de comunicaciones) y *Python* (el lenguaje en el que están implementadas la mayoría de aplicaciones). Los lenguajes de programación soportados son los admitidos por *ICE: C + +, Java, Python, C #, Ruby, PHP y Objective-C*. Los sistemas operativos soportados son *Linux, MacOS X y Windows.*

Microsoft Robotics Studio - http://www.microsoft.com/robotics/

Microsoft Robotics Developer Studio es una plataforma de desarrollo de aplicaciones robóticas basada en Windows. Se trata de una herramienta que permite al usuario implementar las rutinas software que controlarán el comportamiento de su robot, en alguno de los lenguajes de programación soportados por la plataforma .NET, como Visual Basic, C o C#.

Los principales elementos de Robotics Studio son:

- Un lenguaje visual de programación (VPL), que permite la creación intuitiva de aplicaciones para robots (ver figura 2.41).
- Un entorno visual de simulación en tres dimensiones basado en el motor de simulación física AGEIA.
- Soporte en tiempo de ejecución (Runtime) que gestiona la entrada/salida asíncrona, la concurrencia y la distribución de servicios.

El soporte de tiempo de ejecución consta de dos componentes que hacen posible la construcción, supervisión, despliegue y funcionamiento de un gran rango de aplicaciones. Estos dos componentes son el CCR (Concurrency and Coordination Runtime) y el DSS (Decentralized Software Services).

- El CCR permite la coordinación concurrente y asíncrona del flujo de ejecución abstrayendo al programador del uso de hilos, semáforos y otras técnicas de más bajo nivel para el aseguramiento de la exclusión mútua o la prevención del interbloqueo.
- El DSS combina la arquitectura tradicional Web (HTTP) con elementos de las nuevas arquitecturas orientadas a servicios Web (SOAP). La arquitectura resultante está completamente basada en

servicios que se coordinan entre sí para crear aplicaciones distribuidas. El DSS utiliza los protocolos HTTP y DSSP (protocolo propio que ofrece DSS y se encarga de la mensajería entre servicios)

En la figura 2.40 aparece como ejemplo haciendo uso del lenguaje visual de programación VPL, un sencillo diseño para conducir un robot.

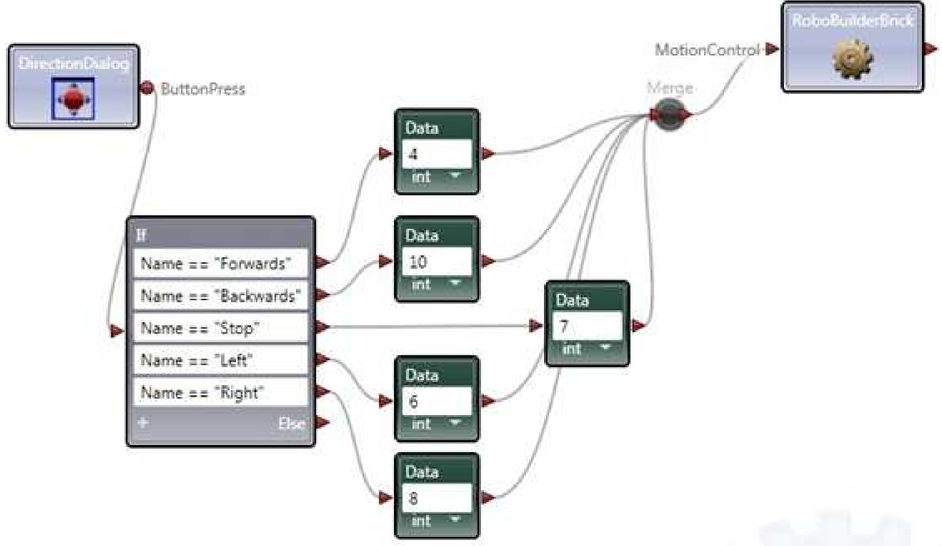
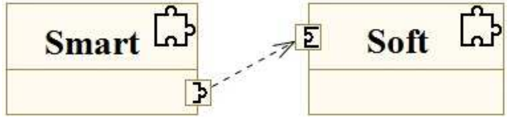


Figura 2.41: Diagrama sencillo VPL para conducir un robot:

SmartSoft - [*http://smart-robotics.sourceforge.net/*](http://smart-robotics.sourceforge.net/)



SmartSoft es un enfoque de componentes para el desarrollo de software robótico, basado en patrones de comunicación como núcleo de un modelo de componentes robóticos. Los patrones de comunicación permiten desacoplar el comportamiento interno y externo del componente.

Para alcanzar los requisitos de los frameworks orientados a componentes se sugiere que todas las interacciones entre componentes se realicen a través de un conjunto predefinido de patrones de comunicación. Esto asegura la disociación de los componentes y elementos relacionados a los niveles apropiados de las abstracciones. La lista de abstracciones que se pueden encontrar en SmartSoft es la siguiente:

- Componentes - Pueden contener varios hilos de ejecución e interactuar con otros componentes a través de los patrones de comunicación.
- Los patrones de comunicación - Definen el modo de comunicación, proporcionar métodos de acceso predefinidos y ocultar toda la comunicación y problemas de sincronización. Siempre es compuesto de dos partes (a) de servicios solicitante y (b) el proveedor de servicios.
- Objetos de comunicación - Representan el contenido que se transmite a través de los patrones de comunicación. Ellos siempre se transmiten por valor. Estos objetos son objetos ordinarios con algunas funciones de miembro adicional.
- Servicio - Cada instancia de un patrón de comunicación proporciona un servicio.

SmartSoft ya ha demostrado su idoneidad como enfoque general de ingeniería de software, incluso fuera de la robótica (Avitrack, 2004). Mientras tanto, el enfoque de la utilización de patrones de comunicación ha sido adoptado por otros grupos, que no sólo proporcionan otro código base sino además demuestran que los patrones de comunicación son un medio adecuado para la introducción de conceptos CBSE en el campo de la robótica (Brooks, 2005).

2.5.3. MDE

Elevando un paso más el nivel de abstracción con respecto a la programación orientada a componentes aparece la Ingeniería Dirigida por Modelos (*Model-Driven Engineering* o MDE). MDE propone la utilización de modelos como concepto principal para el desarrollo de software. Un modelo es una representación simplificada de la realidad que muestra sólo los aspectos que son de interés.

El paradigma basado en modelos ha llamado la atención de la comunidad robótica debido a los resultados muy prometedores que ya ha logrado en otros dominios de aplicación (por ejemplo, la automoción, la aviación, o la electrónica de consumo, entre muchos otros) en términos de mejorar los niveles de reutilización, el aumento de la calidad del software, y la reducción de tiempo de desarrollo. MDE permite a los diseñadores centrarse en los conceptos de dominio, relegando los detalles de implementación a un segundo plano

Un modelo se considera eficaz si tiene sentido desde el punto de vista del usuario y además puede servir como base para la implementación del sistema. Los modelos serán desarrollados a través de una amplia comunicación entre todos los miembros del equipo de desarrollo.

Los modelos se definen en términos formales de meta-modelos, que reúnen los conceptos relativos a un dominio de aplicación particular y las relaciones sintácticas que existen entre estos conceptos (es decir, cada meta-modelo define la sintaxis abstracta de un lenguaje de modelado), así como las normas que determinan cuándo un modelo está bien formado. Al conjunto infinito de todos los modelos válidos que se pueden construir a partir de un meta-modelo se conoce como lenguaje de modelado.

Debido a que un sistema puede ser descrito por diferentes modelos en los distintos niveles de abstracción, las transformaciones de modelos son uno de los temas clave de este enfoque, ya que definen cómo los modelos deben ser interpretados y traducidos. Las transformaciones pueden ser "modelo a modelo" (M2M), y "modelo a texto" (M2T).

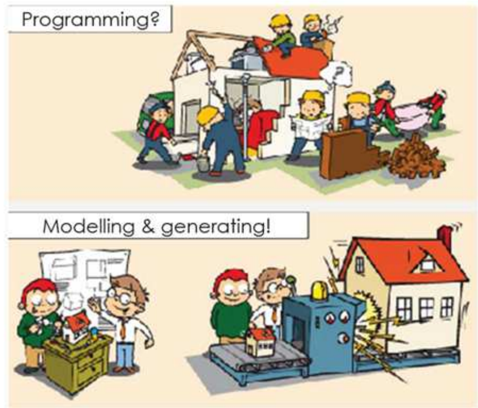


Figura 2.42: Programación tradicional frente al desarrollo dirigido por modelos

Cabe destacar que a pesar de que MDE proporciona una base formal para mejorar el proceso de desarrollo de software, no prescribe la forma más apropiada de seleccionar o diseñar los lenguajes de modelado y de llevar a cabo las transformaciones.

Fruto de esta circunstancia nace MDA (Model-Driven Architecture), que es un acercamiento al diseño de software, propuesto y patrocinado por el OMG (Object Management Group, http://www.omg.org/mda/). Como se muestra en la figura 2.43, los modelos de MDA se pueden clasificar en los siguientes niveles: Modelos de Computación Independiente (CIM), los Modelos Independientes de Plataforma (PIM), y los Modelos Específicos de Plataforma (PSM).

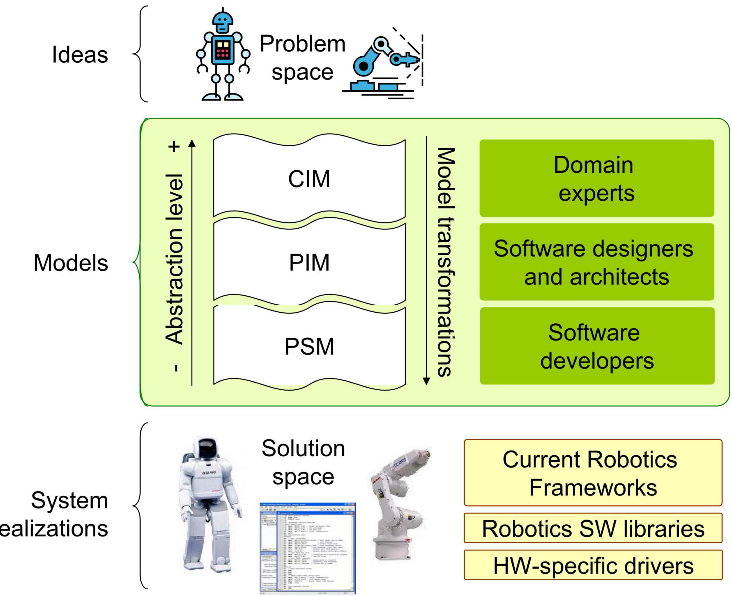


Figura 2.43: MDA en el contexto de desarrollo de software robótico

Dado un modelo independiente de plataforma PIM, puede traducirse a un modelo específico de plataforma PSM para llevar a cabo la implementación correspondiente, usando diferentes lenguajes específicos del dominio o lenguajes de propósito general como Java o C++. Estas 'traducciones' se corresponden a las transformaciones M2M y M2T.

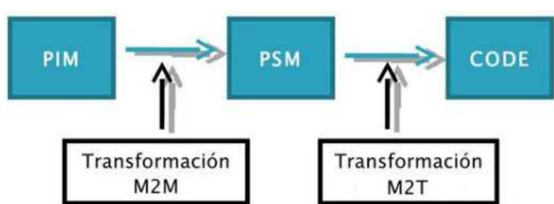


Figura 2.44: Transformaciones M2M y M2T

Conseguir esta separación entre el diseño de la arquitectura y las tecnologías de construcción es uno de los principales objetivos de MDA, ya que facilita que el diseño y la arquitectura puedan ser alterados independientemente. MDA asegura que el modelo independiente de la plataforma (el cual concreta los requerimientos funcionales), sobrevive a los cambios que se produzcan en las tecnologías de fabricación y en las arquitecturas software.

En los últimos años, el enfoque impulsado por MDE) ha demostrado reducir muchas de las limitaciones que se dan en ámbitos donde los sistemas son muy complejos, que necesitan un enfoque de desarrollo multi-disciplinario, y requieren una alta fiabilidad y robustez, como por ejemplo: sistemas integrados [16, 17, 18, 19] la automoción [20], y la automatización del hogar [21, 22]. En el campo de la robótica, el proyecto BRICS [23] está enfocado específicamente a la explotación de MDE como medio que permita reducir el esfuerzo de desarrollo de sistemas de ingeniería robótica, haciendo que las mejores soluciones sean más fácilmente reutilizables.

Otro ejemplo de aplicación "no robótica" que utiliza el desarrollo basado por modelos es WebRatio (www.webratio.com/). WebRatio es una herramienta desarrollada por *WebModels S.L.R*, que proporciona un entorno de desarrollo basado en modelos que permite modelar aplicaciones aplicaciones web, permitiendo la generación automática de código para la plataforma J2EE. Las aplicaciones generadas son aplicaciones JAVA estándar y pueden ser desplegadas en cualquier servidor de aplicaciones Java como Tomcat, JBoss, Resin, IBM Websphere, BEA WebLogic, etc.

WebRatio reduce el tiempo de desarrollo, facilita una rápida presentación de prototipos funcionales y minimizan los costes asociados a desarrollo, implementación y mantenimiento, ampliando la capacidad de los desarrolladores y mejorando la eficiencia de soluciones JAVA desplegables vía web. WebRatio ofrece un editor de diagramas WebML, que permite expresar de manera visual todos los requisitos de la aplicación Web (ver figura 2.45).

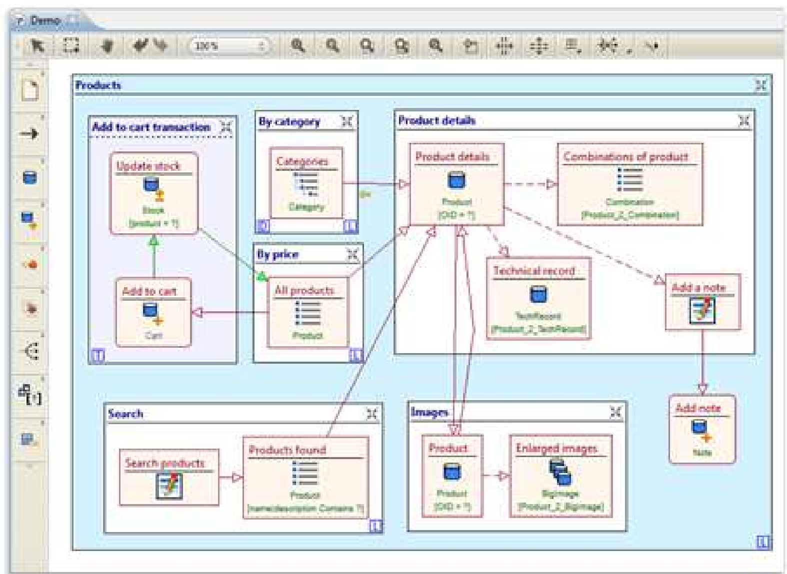


Figura 2.45: Editor de diagramas WebML

Centrándonos ahora en aproximaciones MDE para dominios robóticos podemos presentar V3Studio. El grupo de investigación del DSIE de la Universidad Politécnica de Cartagena, después de sopesar las ventajas y desventajas de UML como principal método de notación, finalmente se decidió a definir un meta-modelo de componentes denominado V3Studio.

Con V3Studio es posible describir la arquitectura de una aplicación sobre la base de sus componentes, y también el comportamiento y los algoritmos que son ejecutados por estos componentes. V3Studio define el conjunto mínimo de elementos necesarios para describir la arquitectura de las aplicaciones, y prescinde de todos aquellos que la experiencia muestra innecesarios.

La figura 2.46 se muestra un esquema del proceso de desarrollo de aplicaciones robóticas utilizando V3Studio. Cada nivel de la pirámide representa un modelo que conforma el meta-modelo ubicado en el nivel superior. El ciclo se cierra en el meta-modelo MOF (Meta Object Facility defined by OMG), que se ajusta a sí mismo.

Como podemos ver en la pirámide de la izquierda, V3Studio es el meta-modelo en el que se construyen los elementos que definen las arquitecturas software. Una vez se genera el modelo V3Studio, puede traducirse directamente en código (transformación M2T en la pirámide de la izquierda), o puede ser traducido en un modelo de expresión orientado a objetos con UML (transformación M2M).

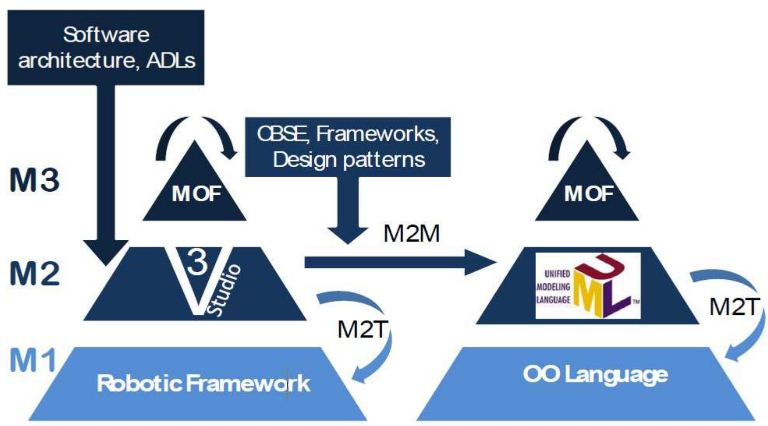


Figura 2.46: Vista general del proceso de desarrollo con V3Studio

Con respecto a SmartSoft, anticipado en el apartado anterior y que profundizaremos en el siguiente capítulo 3; aparte del framework orientado a componentes también posee una herramienta de desarrollo MDE que permite al programador definir y modificar componentes, para posteriormente generar el código que será ejecutado por el framework.

3. SMARTSOFT

3.1. Introducción

SmartSoft es un entorno de programación para aplicaciones robóticas basadas en componentes. Los componentes de SmartSoft están basados en patrones y objetos de comunicación. SmartSoft proporciona un toolchain para el desarrollo de las aplicaciones robóticas denominado *SmartSoftMDSD* que posee un enfoque basado en modelos (*ver apartado MDE- 2.5.3*), así como un framework que aporta un soporte de ejecución. El framework sigue la especificación de middleware de CORBA utilizando la implementación de ACE/TAO para la comunicación entre componentes.

SmartSoft es un enfoque de componentes basado en patrones de comunicación como núcleo de un modelo de componentes robóticos. Las primitivas de la comunicación de SmartSoft son el núcleo de su modelo de componentes. SmartSoft aborda la cuestión de la complejidad proporcionando plantillas de patrones de comunicación estandarizados, que proporcionan una semántica clara de los servicios e impone las interfaces de los componentes implicados.

Dominar la comunicación entre componentes se considera la clave para dominar dependencias y asegurar interfaces uniformes de éstos. El enfoque adoptado por SmartSoft selecciona la comunicación entre componentes como punto de partida en su desarrollo. La idea básica es proporcionar un pequeño conjunto de patrones de comunicación que pueden transmitir objetos entre los componentes y luego exprimir toda interacción entre componentes haciendo uso de estos patrones predefinidos. Un propósito importante de los patrones de comunicación es ocultar en lo posible al constructor del componente los detalles más propensos a errores de los sistemas distribuidos y concurrentes, a través de soluciones aptas y reutilizables. Los patrones de comunicación tienen que proporcionar las pautas para alcanzar un nivel alto de interacción entre componentes.

Ya que todas las interacciones entre los componentes se reducen a los patrones de comunicación predefinidos, las interfaces de los componentes sólo se componen de una misma serie de modelos conocidos con una precisa y predefinida semántica. Esto permite una clara distinción entre el comportamiento externo del componente y su implementación interna, consiguiendo sistemas débilmente acoplados y distribuidos basados en componentes estándar, cuya interacción puede ser ajustada de acuerdo al contexto y necesidades que se den en cada caso, así como garantizar la componibilidad mediante la restricción en la diversidad de estas interfaces (clave en la reutilización de componentes). Los componentes no sólo están desacoplados, sino que también es posible manejar el acceso concurrente dentro de un mismo componente.

Cada componente puede proporcionar y utilizar cualquier número de servicios. Además, el cableado dinámico de los componentes en tiempo de ejecución es apoyado explícitamente por un patrón independiente que interactúa fuertemente con las primitivas de comunicación. En la figura 3.1 aparecen algunos de los elementos más importantes de SmartSoft.

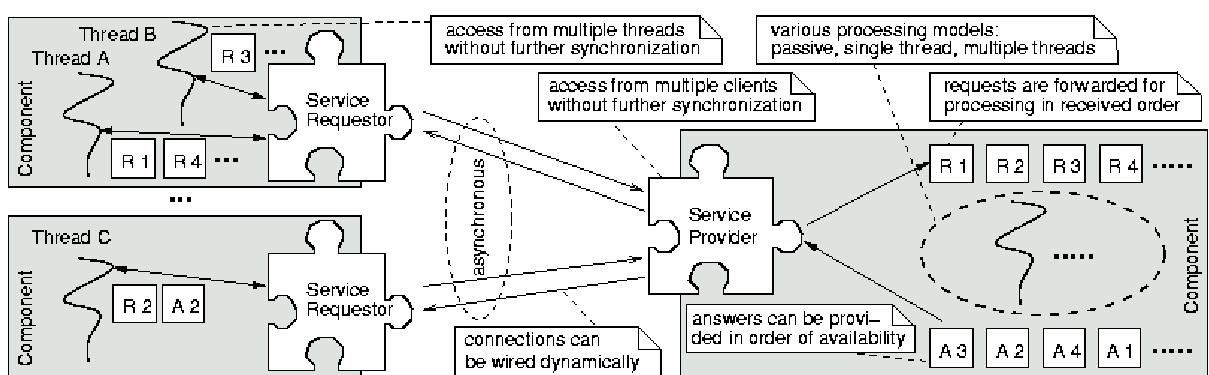


Figura 3.1: Enfoque SmartSoft. El ejemplo muestra el patrón de consulta.

Vemos como un componente proporciona un servicio a través de un puerto de comunicación, a la vez que otros dos componentes poseen un puerto que solicita ese servicio en concreto. Los componentes compartirán entonces los objetos de comunicación a través de dicho canal. Cada componente SmartSoft tendrá un número arbitrario de hilos de ejecución, que serán los encargados de utilizar y gestionar los puertos de comunicación.

Una vez se han construido los componentes así como los objetos de comunicación de los que los primeros hacen uso, en los deployment de SmartSoft se fijarán las comunicaciones entre los componentes generando así la aplicación robótica. El deployment se encargará de convertir los componentes en archivos binarios, para que a gracias al middleware subyacente (CORBA/ACE) se lleve a cabo la comunicación entre ellos.

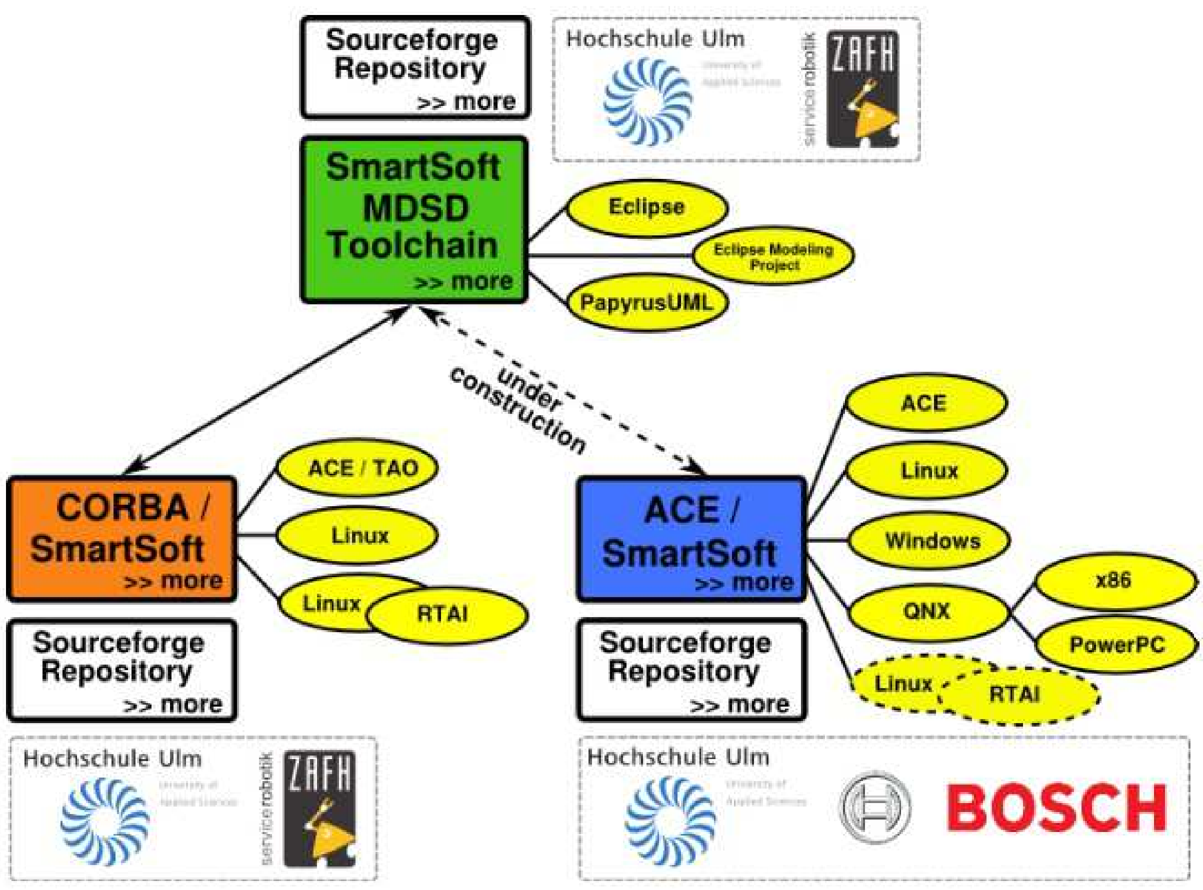


Figura 3.2: El desarrollo de ACE/SmartSoft ha sido apoyado por BOSCH. Corba/SmartSoft y SmartSoft/MDSD se están desarrollando en el Servicerobotik ZAFH de la Universidad de Ulm en Alemania.

En la imagen anterior diferenciamos la herramienta de modelado *SmartSoft MDSD* que a su vez impone el modelo de componentes que define SmartSoft basada en los patrones de comunicación, de las plataformas independientes de ejecución que proporcionan *CORBA/SmartSoft* y *ACE/SmartSoft*. Existen por tanto dos implementaciones del framework de SmartSoft.

La implementación denominada CORBA/SmartSoft hace uso de TAO, que a su vez es una implementación del estándar CORBA haciendo uso del framework ACE para el acceso a sockets (ver apartado *2.3.4 - Middleware*). Por otro lado ACE/SmartSoft prescinde de CORBA y por consiguiente de TAO.), para realizar toda la gestión subyacente de las comunicaciones explotando toda la funcionalidad que aporta ACE.

Si bien es cierto *ACE/SmartSoft* se encuentra en desarrollo, con lo que solo existe una versión funcional de SmartSoft, que es la basada en CORBA (ACE/TAO). Para la realización de PFC siempre se ha hecho uso de esta implementación en concreto.

Sin embargo toda la información aportada por el equipo de desarrollo de SmartSoft está referida a la implementación ACE/SmartSoft, con lo que la documentación del apartado *"3.2. - Elementos fundamentales de SmartSoft"* está de acuerdo a ésta (no así en los siguientes apartados donde se utiliza el MDSD). Aún así se describirá a modo de pie de página, algunos detalles que se consideren importantes sobre la forma en la que se realizan en la implementación CORBA/SmartSoft.

De cualquier modo esto es algo que al usuario final que haga uso de SmartSoft no debe preocuparle, ya que como se comentaba anteriormente la diferencia entre una implementación y otra radica en la plataforma de ejecución que utiliza SmartSoft. La tarea de modelado haciendo uso de SmartSoft MDSD será idéntica en ambos casos. Únicamente advertiremos cambios cuando se analicen detalles de bajo nivel en cuanto a las comunicaciones entre componentes.

Las páginas web de cada una de las implementaciones son:

- *CORBA/SmartSoft* → http://smart-robotics.sourceforge.net/corbaSmartSoft/index.php
- *ACE/SmartSoft* → http://smart-robotics.sourceforge.net/aceSmartSoft/index.php

En el siguiente apartado se describe la herramienta de desarrollo de SmartSoft.

3.2. SmartSoft MDSD Toolchain

SmartSoft sigue una aproximación de Desarrollo Software Dirigido por Modelos (MDSD) (ver apartado *"2.5.4 – MDE"*). SmartSoft es un enfoque de componentes basado en patrones de comunicación como núcleo de un modelo de componentes robóticos que utiliza como middleware CORBA para la comunicación entre componentes, pero que se sirve de una herramienta MDE para el diseño y la modificación de los componentes. De la misma forma, esta herramienta genera el código que ejecuta el framework.

Para llevar a cabo el proceso de implementación del software necesario para nuestra aplicación robótica, se hace necesario una herramienta o cadena de herramientas que liguen todos los conceptos comentados.

Precisamente en informática, un toolchain o cadena de herramientas no es otra cosa que un conjunto de programas informáticos (herramientas) que se usan para crear un determinado producto (normalmente otro programa o sistema informático). Los distintos programas se suelen usar en una cadena, de modo que la salida de cada herramienta sea la entrada de la siguiente, aunque actualmente se abusa del término para referirse a cualquier tipo de herramientas de desarrollo enlazadas.

SmartSoftMDSD es por tanto la herramienta de modelado o toolchain, que impone el modelo de componentes que ha definido el equipo de desarrollo de SmartSoft, y que se sirve del middleware antes mencionado. SmartSoftMDSD se basa en el framework *Eclipse Modeling Project.* Desde hace poco, herramientas como Eclipse han madurado lo suficiente para ser aplicadas y adaptado a la robótica.

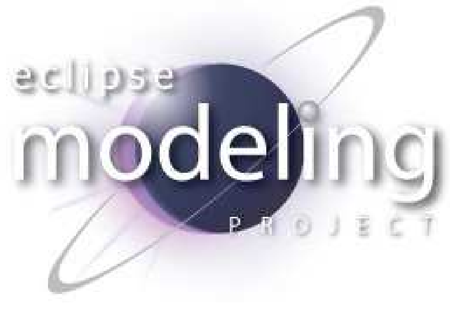


Figura 3.3: Eclipse Modeling Project

Eclipse es un entorno de desarrollo integrado de código abierto multiplataforma. Esta plataforma, típicamente ha sido usada para desarrollar entornos de desarrollo integrados. Sin embargo, Eclipse continúa extendiendo constantemente las áreas de aplicación cubiertas. Fruto de ello es el recientemente creado *Eclipse Modeling Project (EMF,* *http://eclipse.org/modeling/* *)*, que cubre casi todas las áreas de Model Driven Engineering.

EMF es un framework para el modelado y generación de código para construir herramientas y otras aplicaciones basadas en un modelo de datos estructurado.

Desde la especificación de un modelo descrito en *Papyrus* (http://www.papyrusuml.org/scripts/home herramienta gráfica basada en Eclipse para el modelado con UML, de código abierto y ofrecido bajo licencia EPL), EMF proporciona las herramientas y soporte necesario para generar el conjunto de clases acordes al modelo. Java para el modelo, además de clases *adapter* para soportar la visualización y la edición del modelo.

EMF permite la transformación de modelos *Model to Model Transformation (M2M),* consiguiendo de esta manera realizar el proceso de *mapping* y obtener el *PSM (Platform Specific Model)* a partir del *PIM (Platform Independent Model)* que ha sido diseñado mediante *Papyrus*. Además, gracias al denominado *Model to Text Transformation (M2T),* es posible generar el código fuente asociado al modelo.

El metamodelo diseñado gracias *Papyrus (PIM)* es denominado *SmartMARS* (*Smart Modeling and Analysis of Robotics Systems).* Tras hacer uso del *M2M* y realizar el proceso de mapping obtenemos el PSM, o lo que es lo mismo la implementación de SmartSoft basada en la plataforma subyacente como puede ser *CORBA*. Por último gracias al *M2T*, obtenemos los ficheros de fuente que implementan el modelo *PSM,* o lo que es lo mismo el *Platform Specific Implementation (PSI).* Todo este proceso queda reflejado en la figura 3.4.

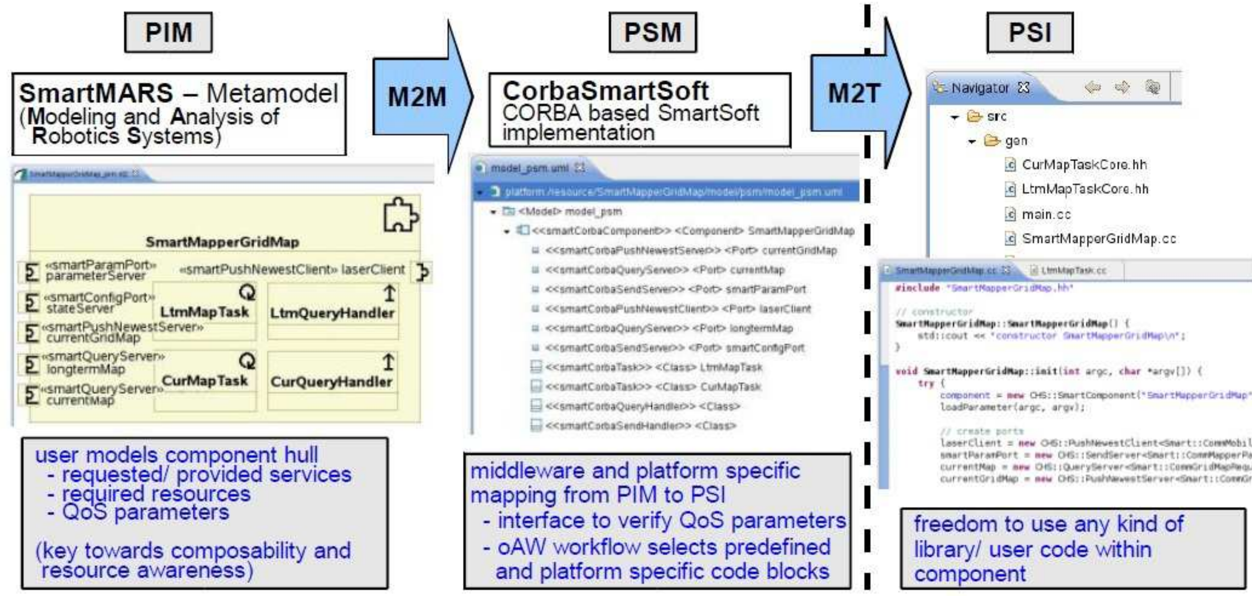


Figura 3.4: Proceso de diseño SmartSoftMDSD

Una vez se han descrito los elementos más importantes de SmartSoft, así como el SmartSoft MDSD y el software subyacente del que este hace uso, en el siguiente capítulo haremos uso de la herramienta de desarrollo indicando de manera precisa la forma de trabajar con ella.

3.3. Elementos fundamentales de SmartSoft

En SmartSoft para construir una aplicación, los componentes deben interconectarse en los deployments, compartiendo objetos de comunicación a través de sus puertos. Entendemos por tanto que los elementos fundamentales de SmartSoft son:

- Objetos de comunicación
- Componentes
- Puertos de comunicación
- Deployments

En los siguientes apartados se profundiza en cada uno de ellos.

3.3.1. Objetos de Comunicación

Figura 3.5: Objetos de comunicación

Los objetos de comunicación son los datos con los que se realiza la comunicación entre componentes. Representan por tanto el contenido transmitido a través de los puertos de comunicación. Siempre se transmiten por valor (equivalente a mover una copia del objeto entre los componentes) para evitar una comunicación *fine grined* entre componentes cuando se accede a un atributo. Los objetos de comunicación son objetos comunes decorados con funciones miembro adicionales.

La genericidad de este enfoque se logra mediante el uso de objetos de comunicación arbitrarios e individuales. El framework transmite sólo el contenido relevante de un objeto de comunicación. En el receptor, ese contenido se utiliza para reconstruir una instancia local del tipo de objeto de comunicación adecuado. Todo el procedimiento es transparente para el usuario y funciona como un constructor de copias (Schlegel, 2004 - apartado 5.4.3)

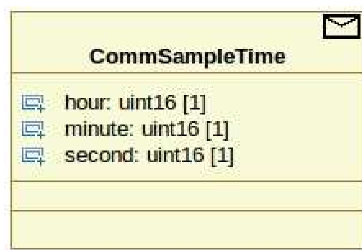


Figura 3.6: Objeto de comunicación CommSampleTime

En la figura 3.6 vemos el modelo del objeto de comunicación *CommSampleTime*, el cual tiene tres atributos todos ellos enteros sin signo de 16 bits: *hour, minute* y *second*. Este objeto de comunicación posee los siguientes ficheros:

** CommSampleTime.hh (Header file for the communication object)*
** CommSampleTime.cc (Implementation of the get(...), set(...) and other methods of the communication object)*

En todo objeto de comunicación podemos diferenciar tres partes: la **estructura interna de datos**, la **interfaz del objeto con el framework**, y **la interfaz de usuario** del objeto. A continuación veremos cada una de ellas por separado.

1) Estructura de datos

El primer paso para crear un objeto de comunicación es decidir la estructura de los datos que intercambiarán los componentes. Es conveniente utilizar un formato de datos canónico (por ejemplo las unidades físicas del Sistema Internacional, como metro, segundo...). Este es uno de los pasos más importantes, ya que tiene consecuencias en la interfaz de usuario. Estos datos sin procesar son la representación de los datos internos del objeto de comunicación. A pesar de que pueden consistir de tipos de datos simples (como *int, float...*) esto no es recomendable.

La razón es que ACE realiza automáticamente las conversiones de tipos de datos, que puede conducir a suposiciones erróneas acerca del tipo de datos y la longitud del tipo. La mejor manera es utilizar ACE_CDR (Common Data Representation) y los tipos de datos definidos en la librería de ACE (ver Figura 3.5). Esta es la mejor manera de definir tipos de datos uniformes, para que se puedan convertir desde/a cualquier de representación de datos dependiente de plataforma (estos tipos de datos también se utilizan en el ACE-TAO (CORBA) – implementación IDL). Además de los tipos de datos ACE_CDR, los tipos de datos sin procesar (raw data) pueden consistir en cualquier tipo de estructura en forma de contenedores STL (por ejemplo iteradores, listas, vectores, etc)

ACE_CDR Type-Name	*Size in bits*	*typical typedef on 32 bit system*
ACE_CDR::Boolean	8	bool
ACE_CDR::Octet	8	unsigned char
ACE_CDR::Short	16	ACE_INT16
ACE_CDR::UShort	16	ACE_UINT16
ACE_CDR::Long	32	ACE_INT32
ACE_CDR::ULong	32	ACE_UINT32
ACE_CDR::LongLong	64	platform specific
ACE_CDR::ULongLong	64	platform specific
ACE_CDR::Float	32	platform specific
ACE_CDR::Double	64	platform specific
ACE_CDR::LongDouble	128	platform specific

Figura 3.7: ACE_CDR data types

La mejor manera de crear una representación interna de datos es definir una estructura que pueda ser almacenada en un archivo de encabezado independiente. Esto dará lugar a una estructura similar a la de los archivos CORBA-IDL.

Como ya hemos comentado, a partir de estos tipos de datos básicos es posible componer estructuras de datos de gran complejidad para los objetos de comunicación. En la figura 3.6 se presenta una serie de estructuras de datos que pertenecerían a un objeto de comunicación utilizado en la lectura del láser.
La estructura de datos denominada *StructLaserScan*, además de tener tipos de datos *Boolean, UShort* y *ULong*; también posee una variable denominada *time* que a su vez es una estructura de datos del tipo *StructTimeStamp*. También posee un contenedor de tipo lista denominado scan que almacena tipos *StructScanPoint.*

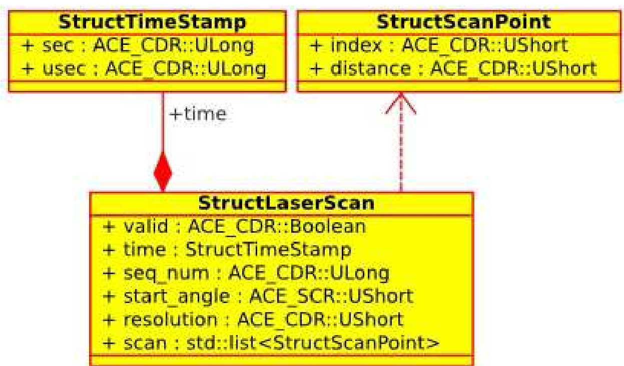


Figura 3.8: Estructura de datos del objeto LaserScan

2) Framework interface

Desde el punto de vista del framework, en la implementación ACE/SmartSoft, un objeto de comunicación es una clase que tendrá al menos las siguientes tres funciones [1]:

```
-   void get(ACE_Message_Block *&msg) const;
-   void set(const ACE_Message_Block *msg);
-   static std::string identifier(void);
```

Como ya se ha comentado, los objetos de comunicación son objetos normales que están decorados por las funciones miembro adicionales para su uso por el framework. El método *get* extrae las estructuras de datos relevantes y las convierte en una representación independiente de plataforma para la transmisión. El método *set* convierte la representación independiente de plataforma de nuevo en la representación interna del objeto original. Finalmente, cada tipo de objeto de comunicación puede ser identificado por un nombre único proporcionado por el método identificador (Schlegel, 2004 - apartado 5.4.3).

En general la interfaz del framework es responsable de convertir la estructura interna de datos en/desde lo que se denomina CDR-Stream (CDR: Common Data Representation). Para ello la librería de ACE ofrece dos clases, denominadas ACE_OutputCDR y ACE_InputCDR. El ACE_Message_Block almacena los datos de manera uniforme respetando la alineación de datos (tipos de datos y los tamaños) y el cálculo de referencias (orden de bytes).

Esta conversión de datos se lleva a cabo en los métodos *get(...)* y *set (...)*, tal como se ha descrito al principio). Además, el método de identificación *identifier(...)* se utiliza para diferenciar el objeto de comunicación de otros. Para ello, el método de identificación debe implementarse de tal forma que siempre devuelva el nombre de un objeto de comunicación (*string*) que es único entre todos los objetos de comunicación en un sistema.

Los métodos **identifier***(...)***, get***(...)*, y **set***(...)* son **generados automáticamente por el MDSD** de SmartSoft en el proceso de generación del código asociado al modelo en el diseño del objeto de comunicación.

En la figura 3.9 se muestra el objeto de comunicación *CommLaserScan*. Además de la estructura de datos denominada *StructLaserScan*, vemos como el objeto de comunicación posee las tres funciones antes mencionadas. Por razones de simplicidad, se ignora por un lado la interfaz de usuario (que será descrita más adelante) y se simplifica la estructura de datos *StructLaserScan* (que se describió en el apartado anterior).

1- En CORBA/SmartSoft esas funciones quedan de la siguiente manera:
- *void get (CORBA::Any &a) const;*
- *void set (const CORBA::Any &a);*
- *static std::string indentifier (void);*

Figura 3.9: Interfaz del framework para el Objeto de Comunicación de un barrido de láser LaserScan

La implementación del método *identifier (...)* queda de la siguiente manera:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
std::string CommLaserScan::identifier(){
      return "CommLaserScan";
}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Como se muestra en la imagen, el barrido del láser consiste en un ángulo de partida, una resolución y los puntos de análisis (se almacena como una lista de las estructuras punto-scan). Los métodos *get(...)* y *set(...)* quedan implementados de la siguiente forma:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
void CommLaserScan::get(ACE_Message_Block *&data) const{
      // define a local output-stream with default beginning-size of 512k
      ACE_OutputCDR out(ACE_DEFAULT_CDR_BUFSIZE);

      // 1) first we store the direct members
      out << laser_scan.start_angle;
      out << laser_scan.resolution;

      // 2) then we store the list items

      // 2.1) the number of elements in list has to be saved
      ACE_CDR::ULong size = laser_scan.scan.size();
      out << size;

      // 2.2) save all list items
      std::list::const_iterator iter;

      for(iter=laser_scan.scan.begin(); iter != laser_scan.scan.end(); iter++){
            out << iter->index;
            out << iter->distance;
      }

      // finally the stream is returned (as a copy) in the data parameter
      data = out.begin()->clone();
}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

El método *set (...)*:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
void CommLaserScan::set(const ACE_Message_Block *data){
      // get the stream out of message-block
      ACE_InputCDR input(data);

      // 1) first we read the direct members
      input >> laser_scan.start_angle;
      input >> laser_scan.resolution;


      // 2) then we read the list items

      // 2.1) the number of elements in list has to be restored
```

```
        ACE_CDR::ULong size;
        input >> size;

        laser_scan.scan.clear();

        // 2.2) read all list items
        StructScanPoint scan_point;
        for(ACE_CDR::ULong i=0; i < size; i++){
                input >> scan_point.index;
                input >> scan_point.distance;

                laser_scan.scan.push_back(scan_point);
        }

        // data message-block will be cleared automatically
}
```
%%%

A partir de aquí el objeto de comunicación se puede utilizar en *ACE/SmartSoft*. Ya únicamente queda la interfaz de usuario que define la conexión con el código de usuario.

3) User interface

Una vez que la estructura interna de datos y la interfaz de framework están definidas, se puede implementar la interfaz de usuario. La interfaz de usuario se compone de funciones '*getter'* y '*setter'* para leer/modificar la estructura de datos interna. Este es un paso importante en el desarrollo, porque la asignación de los tipos de datos del usuario a los tipos uniformes ACE_CDR (y viceversa) se lleva a cabo aquí.

En la mayoría de los casos, es una asignación directa (por ejemplo: *int to ACE_CDR::Long*, o *double to ACE_CDR::Double*). Una vez más se utiliza el ejemplo del objeto de comunicación LaserScan que esta vez se enriquecerá con la interfaz de usuario.

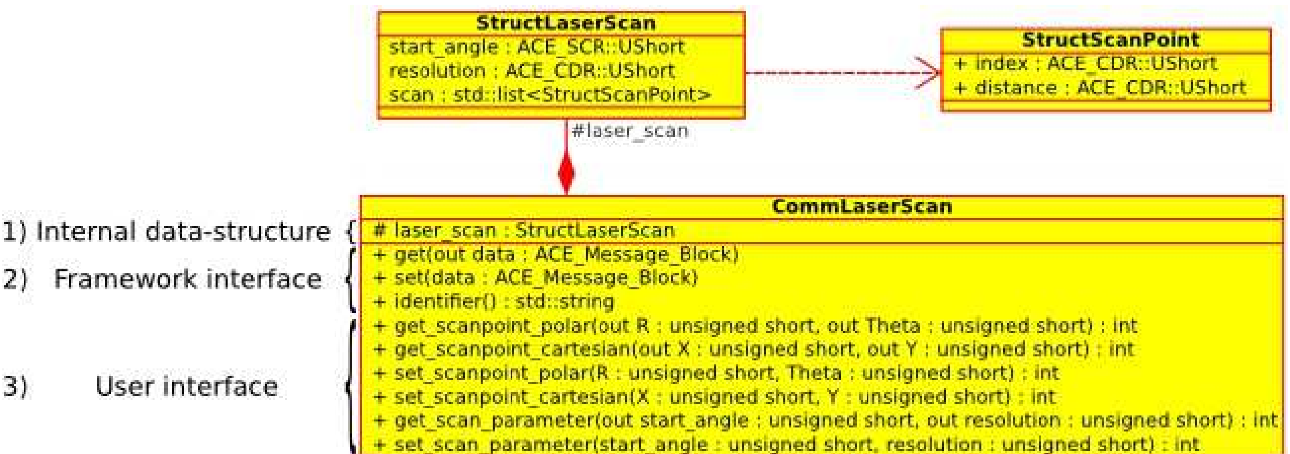


Figura 3.10: Objeto de comunicación completo con interfaz de usuario.

Por tanto, la interfaz de usuario estará formada por varias funciones '*setter'* y '*getter'* que ofrecen un acceso especificado por el usuario a los datos internos del objeto de comunicación. Por ejemplo, el usuario puede elegir entre un barrido láser en coordenadas cartesianas o polares. Los datos fundamentales siguen siendo los mismos porque la conversión se puede hacer en línea en las funciones *'getter/setter'*. Por lo tanto la responsabilidad de las funciones '*setter'* y '*getter'* es convertir los datos en varios formatos o semántica (que el usuario pueda necesitar). Por supuesto lo óptimo es reducir las conversiones tanto como sea posible (que es a menudo el caso con el modelo de datos canónico).

La implementación de las funciones miembro de usuario de los objetos de comunicación, no se ven afectadas por aspectos de la comunicación entre componentes ni pueden introducir dependencias entre componentes.

De acuerdo con la visión basada en servicios, los objetos de comunicación pueden ser accedidos y manipulados sin que ello suponga interacciones externas de componentes, ya que su contexto de ejecución nunca se extiende a todos los componentes. También pueden ser utilizados independientemente del estado posterior del componente que proporciona el servicio, una vez que se ha obtenido el objeto de comunicación.

3.2.2. Componentes

Los componentes son la base de SmartSoft como unidades reutilizables que interaccionan entre ellos intercambiando objetos de comunicación por sus puertos, ofreciendo/solicitando un servicio determinado. En SmartSoft cada componente se implementa como un proceso, que puede contener varios hilos de ejecución. Los componentes pueden ser conectados de forma dinámica en tiempo de ejecución. En la figura 3.9 vemos como los componentes se comunican unos con otros, a través de los servicios prestados/requeridos de sus puertos de comunicación.

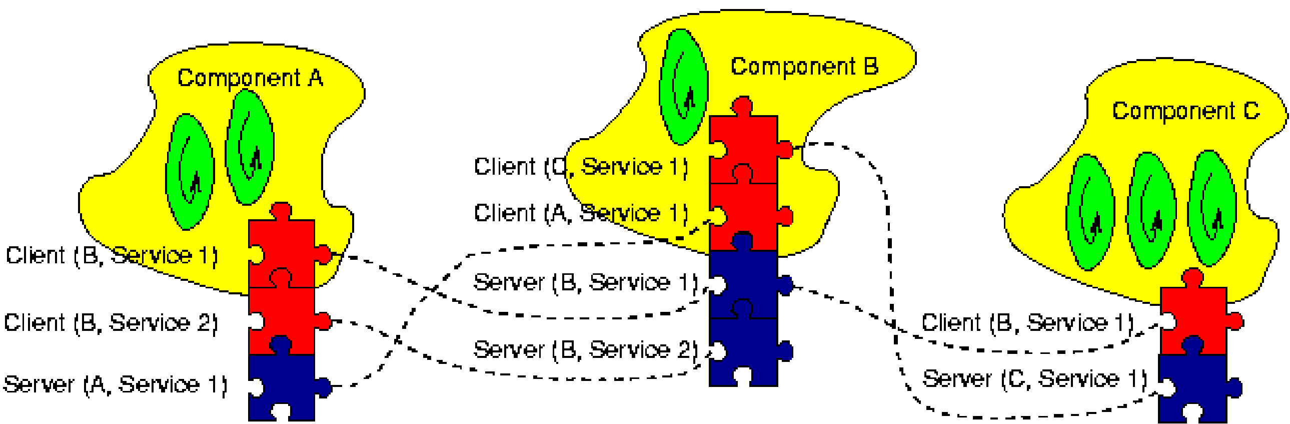


Figura 3.11: Comunicación entre componentes SmartSoft

El enfoque propuesto sostiene una comunicación entre componentes a nivel de servicios, en lugar de realizarlo mediante funciones miembro, lo que daría lugar a una comunicación de tipo "*fine grained*" (cantidad de cómputo con relación a la comunicación). Puesto que todos los servicios que ofertan y requieren los componentes se basan únicamente en un conjunto muy reducido de patrones, el diseñador de componentes puede controlar de manera estricta las dependencias entre componentes y asegurar de esta manera que las interfaces entre componentes están bien formuladas.

El "casco" (hull) del componente separa la parte interna del componente, de la visión externa (servicios ofrecidos y requeridos). Esto se logra por medio de los patrones de comunicación predefinidos en SmartSoft.

El modelo de componentes de SmartSoft incluye un autómata interno de estados que representa al menos los estados *neutral, alive y fatal*. Además, el casco del componente proporciona un contenedor de ejecución (un entorno de ejecución) que ofrece servicios normalizados del sistema operativo como hilos de ejecución (threads), timers, o mutex; independientemente de la tecnología de implementación subyacente.

En la figura 3.10, vemos como se hace distinción entre la parte interna y externa del componente, así como los elementos que entran en juego en cada una de ellas.

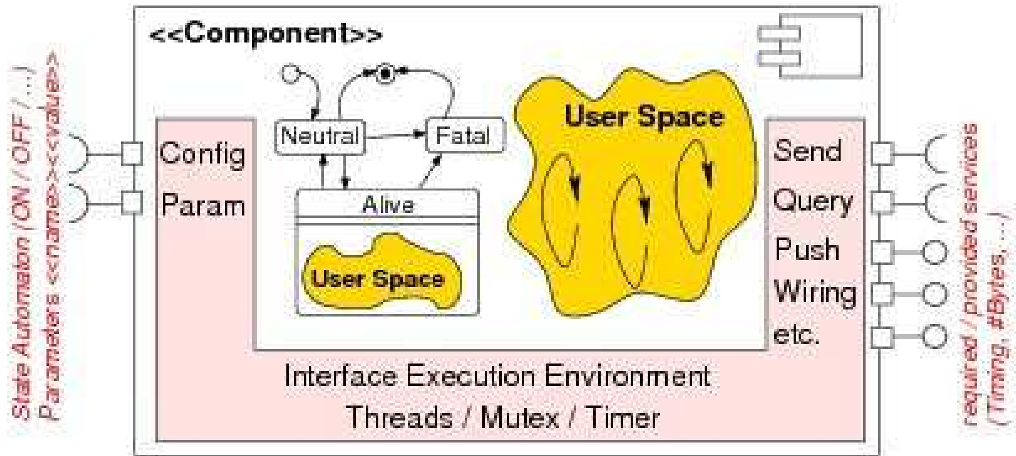


Figura 3.12: Componente SmartSoft

Resulta interesante separar los dos puntos de vista diferentes de un componente y su modelo de comunicación subyacente:

**** Vista externa del componente***

- Desde fuera de un componente otro componente solo ve sus puertos de servicio. La semántica del puerto de servicio es completamente especificada por el modelo de comunicación subyacente y sus objetos de comunicación.
- Esto es suficiente para entender completamente el comportamiento del servicio y saber cómo ponerse en contacto con él.
- Este punto de vista, considera el componente como una *caja negra.* Su funcionamiento interno y la implementación de un servicio no son relevantes ya que los patrones de comunicación muestran una semántica estandarizada de puertos.
- Desde el punto de vista externo no se requiere comprender los mecanismos de comunicación que existen por debajo de los patrones de comunicación, siendo éstos los elementos de mayor interés.

**** Vista interna del componente***

- El implementador del componente se centra en procesamiento de los datos, realizando la parte algorítmica propia de éste.
- Los datos con los que trabaja el componente serán objetos de comunicación. Estos objetos pueden proceder de otros componentes y haber sido recibidos a través de los puertos de comunicación.
- De la misma manera, es posible enviar un objeto ya tratado a través de los distintos servicios proporcionados por el componente.
- Desde este punto de vista son de especial interés elementos como los Handlers o Tareas; por ser los encargados de realizar esas tareas de procesamiento antes mencionadas.

Esta separación tan clara entre vista o capas externa e interna y las actividades que se desempeñan en cada una de ellas, está apoyada por el software subyacente como es ACE/TAO. De esta manera, SmartSoft aporta al usuario facilidades para que no tenga que preocuparse de ciertos detalles de bajo nivel como pueden ser la inicialización y parada de componentes, establecimiento de las comunicaciones, manejo de excepciones en caso de error… El casco del componente encapsula el área de código definido por el usuario, a la vez que lo aísla de detalles internos específicos.

Además de los puertos de comunicación (ver *apartado 3.2.3 – puertos de comunicación*), SmartSoft proporciona un conjunto de elementos con los que realizar el modelo e implementación de los distintos componentes.

Las tareas SmartTask dan al usuario la posibilidad de implementar tareas concurrentes portables a diferentes plataformas siempre y cuando el usuario no añada código específico del sistema operativo. Para llevar a cabo la implementación de las tareas de usuario es posible utilizar las clases SmartTask o ManagedTask. SmartTask es una implementación simple de un objeto activo utilizando ACE_Task como base. ManagedTask es similar a SmartTask con la diferencia de que añade la gestión centralizada del hilo. Así, la creación y aborto del hilo se manejan dentro del *ACE/SmartSoft Kernel.* Esto ofrece al usuario la opción de ser informados sobre el aborto del hilo en el interior de un ManagedTask

Figura 3.13: Modelo de una tarea SmartTask

Por otro lado los Handlers son utilizados por los componentes para realizar la gestión de los puertos de comunicación. Como veremos más adelante, algunos patrones de comunicación llevan asociados estos handlers o manejadores, de forma que son lanzados por el software subyacente si así lo impone el tipo de patrón. Su uso o finalidad difiere del que tienen las tareas SmartTask, ya que mientras las tareas son a priori las encargadas de realizar el procesamiento en sí del componente, los Handlers son encargados de realizar actividades concretas (con poco procesamiento en general) en momentos determinados.

Figura 3.14: Modelo de un Handler

Otros elementos de interés son los *Mutex*. Las variables *Mutex* son la forma más común de implementar la sincronización de hilos y de proteger datos compartidos cuando son compartidos por varios hilos de ejecución.

Figura 3.15: Modelo de un SmartMutex

También se ofrecen temporizadores SmartTimer, que permiten ejecutar un determinado segmento de código, cada cierto intervalo de tiempo que el usuario haya definido previamente.

Figura 3.16: Modelo de un SmartTimer

En el proceso de modelado del componente también es posible establecer variables de inicialización o configuración en el modelo, de manera que sean visibles y compartidas por todas las tareas o handlers que el componente posea, pudiendo modificar más tarde su valor. Estas variables pueden ser desde un simple tipo entero, un objeto de comunicación, o un objeto de una clase cualquiera definida por el usuario. Estas variables al generar el código asociado al modelo son definidas en el fichero *".ini"* asociado al proyecto del componente.

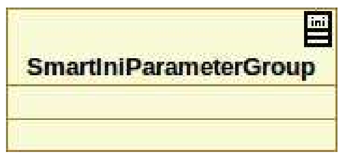


Figura 3.17: Modelo de un SmartIniParameterGroup

A continuación se muestra un ejemplo donde mostrar cómo algunos de los elementos antes mencionados forman parte de un componente en concreto. En la figura 3.16 vemos el modelo de un componente denominado LaserRangeFincher realizado con el MDSD de Smartsoft. El comportamiento que tiene dicho componente y como lo definen las tareas y handlers, los patrones que implementan los puertos de comunicación, etc… son detalles irrelevantes en estos momentos, que se profundizará en posteriores apartados.

El componente posee una tarea *LaserTask* que gestiona dos puertos de comunicación: *BaseStatePushNewestClient* y *LaserPushNewestServer*. Por otro lado existe un puerto *LaserQueryServer* que tiene asociado un handler *LaserQueryHandler.* Por último vemos como existe un *SmartIniParameterGroup* denominado *LocalLaserScan* que contiene un objeto de comunicación del tipo *CommMobileLaserScan* que es compartido y utilizado tanto por la tarea como por el handler.

Destacar que las flechas que aparecen no forman parte del modelo de SmartSoft, sino que únicamente aparecen para una mejor descripción de la figura. Más adelante veremos cómo se relacionan unos elementos a otros.

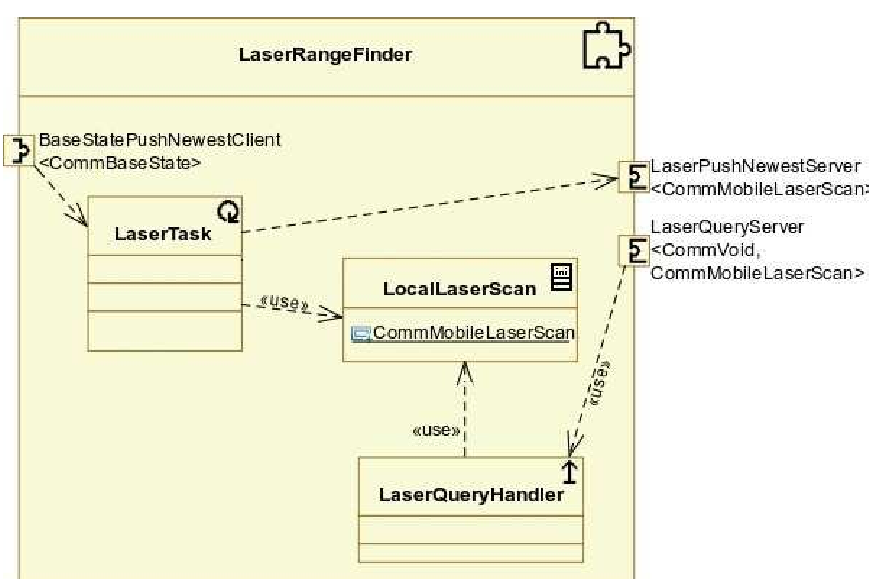


Figura 3.18: Ejemplo de Componente en SmartSoft

El primer paso en la ejecución de un componente ACE/SmartSoft es inicializar su casco. Este paso generalmente se realiza una vez al principio y después todo queda estable. Este código es generado automáticamente por el MDSD de SmartSoft a la hora de realizar el proceso de generación de código asociado al modelo a la hora de diseñar un componente.

Vamos a analizar el código generado para un componente cualquiera. Tomamos como ejemplo el componente *SmartSampleSend* que analizaremos en siguientes apartados. El que se haya escogido este componente en concreto, o el comportamiento específico que posee es algo irrelevante en estos momentos, ya que el listado de código que aparece a continuación necesario para inicializar un componente, será idéntico en todos los casos.

Por defecto SmartSoft asigna al proyecto el mismo nombre que el componente. De esta manera, en el fichero *main.cc* de la carpeta *src/gen* del proyecto asociado tenemos lo siguiente:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#include "SampleSendClient.hh"

SampleSendClient SampleSendClient::_sampleSendClient;

int main(int argc, char *argv[]){
       std::cout << "main...\n";

       SampleSendClient::instance()->init(argc, argv);
       SampleSendClient::instance()->run();

}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Por lo tanto vemos como el proceso principal de todo componente es la invocación de las funciones *init (…)* y *run (…)* propias del propio componente. La función init (…) la encontramos en el fichero ".cc" con mismo nombre que el proyecto (*SmartSampleSend.cc* para este caso en concreto) de la carpeta *src/gen, y* contiene lo siguiente:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
void SampleSendClient::init(int argc, char *argv[]){
   try{
       component = new CHS::SmartComponent("SampleSendClient", argc, argv);

       // create ports
       printClient = new CHS::SendClient<CommSampleObjects::CommSamplePrint>(component);

       } catch (const CORBA::Exception &){
              std::cerr << "Uncaught CORBA exception" << std::endl;
       } catch (...){
              std::cerr << "Uncaught exception" << std::endl;
       }
}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

La primera parte notable de este listado de código es el bloque *try-catch* alrededor del *SmartComponent. ACE/SmartSoft* tiene una política estricta en cuanto a excepciones. Las excepciones podrían ser lanzadas sólo durante la inicialización. No serán lanzadas en tiempo de ejecución, en lugar *ACE/SmartSoft* proporciona valores de retorno *StatusCode* para ese fin (véase el *Enumerations declaration* en el Doxygen para más detalles).

A continuación se realiza una instancia de un SmartComponent con el nombre del componente y los principales parámetros estándar. Igual ocurre con los puertos de comunicación que posea el componente.

El método *run (…)* se encarga de iniciar todos los mecanismos internos del componente de *ACE/SmartSoft:*

- El NamingService se pondrá en marcha
  - Se buscará un archivo de configuración (por lo general *cfg.conf*) que parametriza el cliente interno del *NamingService*
  - El cliente del *NamingService* se pondrá en marcha (el demonio del *NamingServcie* tiene que estar ejecutando)
- Todos los mecanismos internos para manejar los puertos de comunicación, las tareas del usuario, *timing* y estados internos se iniciarán.
- El bucle del hilo principal se utiliza para el ciclo interno del componente que se sigue ejecutando mientras no se interrumpa (por ejemplo, con STRG +C *command)*

3.3.3. Puertos de comunicación

Los **puertos de comunicación** ayudan al constructor de componentes y al constructor de aplicaciones en la creación y uso de componentes distribuidos de tal manera que la semántica de la interfaz está predefinida por los **patrones de comunicación**, independientemente del lugar en que se aplican. Un patrón de comunicación define el modo de comunicación, proporciona métodos de acceso predefinidos y oculta todos los problemas de comunicación y sincronización. Un puerto de comunicación siempre se compone de dos partes complementarias llamadas solicitante de servicio y proveedor de servicio que representan una relación cliente/servidor, maestro/esclavo, o publicador/suscriptor.

Los patrones de comunicación son un pequeño conjunto de plantillas para los puertos de componentes con una semántica de interacción estrictamente definida. Los puertos de los componentes pueden definirse únicamente sobre la base de estos patrones de comunicación predefinidos.

Cada instancia de un patrón de comunicación proporciona un **servicio.** Un servicio comprende el modo de comunicación tal como se define por el patrón de comunicación y el contenido definido por los objetos de comunicación. El conjunto de los patrones de comunicación está resumido en la figura 3.19. Aunque en el desarrollo de componentes se habla siempre de servicios ofrecidos y requeridos, SmartSoft se refiere a puertos cliente y servidor.

Patrón	Relación	Comunicación
Send	*Cliente/Servidor*	*Unidireccional*
Query	*Cliente/Servidor*	*Bidireccional*
Push newest	*Publicador/Suscriptor*	*Distribución '1' a 'n'*
Push timed	*Publicador/Suscriptor*	*Distribución '1' a 'n'*
Event	*Cliente/Servidor*	*Notificación asíncrona*
Wiring	*Maestro/Esclavo*	*Cableado dinámico*
State	*Maestro/Esclavo*	*Gestión del estado*

Figura 3.19: Conjunto de patrones de comunicación de SmartSoft

Los patrones de comunicación realizan varios modos de comunicación, como puede ser la comunicación de un solo sentido o unidireccional que utiliza el patrón de comunicación “Send”; o una interacción pregunta/respuesta o bidireccional como es el caso del patrón “Query”. El proveedor del servicio podrá manejar cualquier número de clientes al mismo tiempo.

Los servicios “Push” son proporcionados por el “Push Newest” y por el “Push Timed”. Mientras que “Push Newest” puede ser utilizado para distribuir datos irregularmente a los clientes suscritos siempre que haya actualizaciones disponibles, el patrón “Push Timed” distribuye los datos regularmente con el intervalo de tiempo que el constructor del componente haya definido.

Al igual que ocurría con los patrones “Send” y “Query”, el proveedor del servicio también podrá manejar un número arbitrario de clientes.

El patrón de comunicación “Event” es utilizado para la notificación asíncrona si la condición de activación de un evento se hace true bajo los parámetros de activación. El patrón “Wiring” incluye el cableado dinámico de los componentes en tiempo de ejecución. Por último, el patrón de comunicación “State” proporciona un mecanismo para conocer el estado del componente. El funcionamiento de cada uno de los distintos patrones de comunicación se verá por separado en los siguientes apartados. Debido a la gran importancia que tienen los patrones de comunicación, profundizaremos en cada uno de ellos en siguientes apartados.

El conjunto de patrones de comunicación no es el más pequeño posible, ya que una comunicación asíncrona de un solo sentido sería suficiente para poner en práctica cualquier modalidad de comunicación. Restringir todas las interacciones entre componentes a los patrones de comunicación requiere un conjunto de patrones que sea lo suficientemente completo para cubrir todas las necesidades de comunicación. Por supuesto, también se desea encontrar el conjunto más pequeño posible a fin de conseguir una máxima claridad en las interfaces de los componentes y evitar esfuerzos innecesarios en las implementaciones de los patrones de comunicación. Por tanto el equipo de desarrollo del framework SmartSoft ha intentado mantener el número de patrones de comunicación lo más pequeño posible, sin restringir el uso fácil.

Las funciones miembro predefinidas de los patrones proporcionan distintos modos de acceso como la invocación de servicios de manera síncrona y asíncrona, o un controlador basado en el manejo de peticiones. En algunas ocasiones si bien es cierto no se permitirá al constructor de componentes decidir sobre la conveniencia de llamar a un servicio remoto de forma síncrona o asíncrona para un patrón en concreto, pero así se consigue una facilidad en el uso y se aporta un conjunto fijo de modos de acceso para cada patrón de comunicación. En particular, esto ofrece la oportunidad de manejar de forma totalmente transparente al usuario la concurrencia y el asincronismo dentro de los patrones de comunicación, en lugar de ocuparse de estas cuestiones el constructor una y otra vez cada vez que define un objeto visible en la interfaz del componente.

Los patrones de comunicación ocultan el middleware subyacente y no esperan que el usuario del framework por ejemplo se enfrente con detalles de CORBA. En comparación con los objetos de comunicación, en los patrones de comunicación el usuario no puede exponer arbitrariamente funciones miembro como interface de los componentes, ni puede diluir la estricta semántica de la interfaz. De esta manera las dos partes que se comunican evitan acabar dándole vueltas a la semántica de las interfaces de los componentes.

Dado que los objetos de comunicación siempre se transmiten por los puertos de comunicación y que las funciones miembro de los objetos de comunicación no están expuestas fuera de un componente, el uso de objetos de comunicación y la implementación de las funciones miembro de usuario se encuentran libres de incómodos y complicados detalles en cuanto a mecanismos comunicación entre componentes y objetos distribuidos.

Dependiendo del rol que tenga el componente en el proceso de comunicación, para un mismo patrón de comunicación tendremos una API de usuario u otra. Los métodos de acceso de un patrón de comunicación desde el interior del componente sólo son visibles desde esta posición, no siéndolo desde el exterior y no formando parte del servicio que proporciona el componente.

De igual manera ocurre con los métodos de acceso del patrón de comunicación cuando el componente hace las veces de cliente. Cuando se estudien los distintos patrones de comunicación se observarán las diferentes APIs de las que dispone el componente según sea su rol (cliente/servidor). En la figura 3.18 se aprecia las distintas APIs que tendrán los componentes para el patrón de comunicación "PushTimed", según sea el rol que desempeñan.

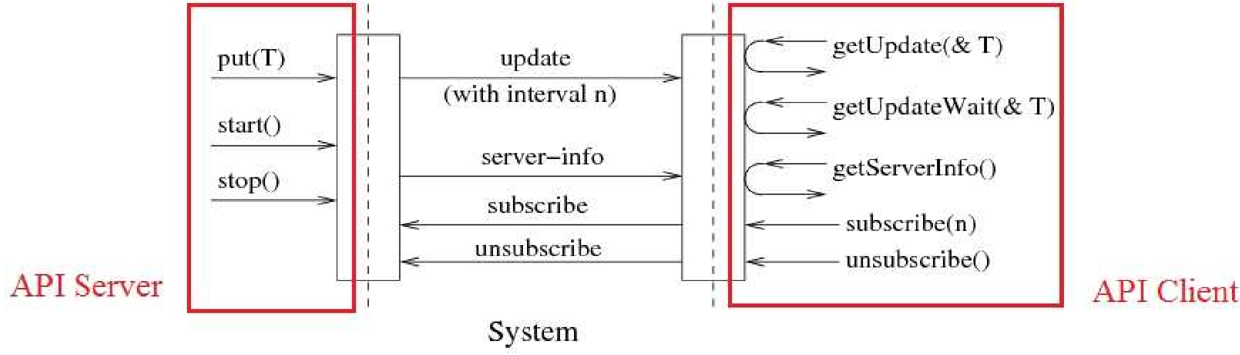


Figura 3.20: Distintas API's de los puertos para el proveedor y suscriptor del patrón PushTimed

3.3.4. Deployment

Un deployment software es el conjunto de todas las actividades necesarias para que un sistema software quede disponible para su uso.

El proceso de deployment en general se compone de varias actividades interrelacionadas con posibles transiciones entre ellas. Debido a que cada sistema software es único, los procesos o procedimientos precisos de cada actividad no se pueden definir. Por lo tanto, ese deployment o despliegue se debe interpretar como un proceso general que ha de ser personalizado de acuerdo a las necesidades o características específicas.

Un deployment en SmartSoft es un proyecto en el que diseñar una aplicación. En el deployment se hace uso de los componentes, estableciendo los canales de comunicación que existirán entre los éstos a través de los puertos de comunicación. Por tanto el deployment es ese lugar donde componer o ensamblar las distintas "piezas del puzle". Los encargados de realizar los deployment son los constructores de aplicaciones, que realizarán o montarán aplicaciones basadas en componentes anteriormente construidos (Ver apartado 2.6).

Los deployment se encargan de convertir los componentes que forman parte de la aplicación en archivos binarios, para que a gracias a middleware subyacente como CORBA/ACE se lleve a cabo la comunicación entre ellos. A continuación se muestra un ejemplo de deployment para una tarea de navegación de un robot.

El *Base Server* y el *Laser Server* son componentes de acceso al hardware específico, que proporcionan acceso a una plataforma robótica y láser. Los componentes de localización del robot, la construcción de mapas, planificación de trayectorias y la ejecución de movimiento implementan funcionalidades de navegación del robot.

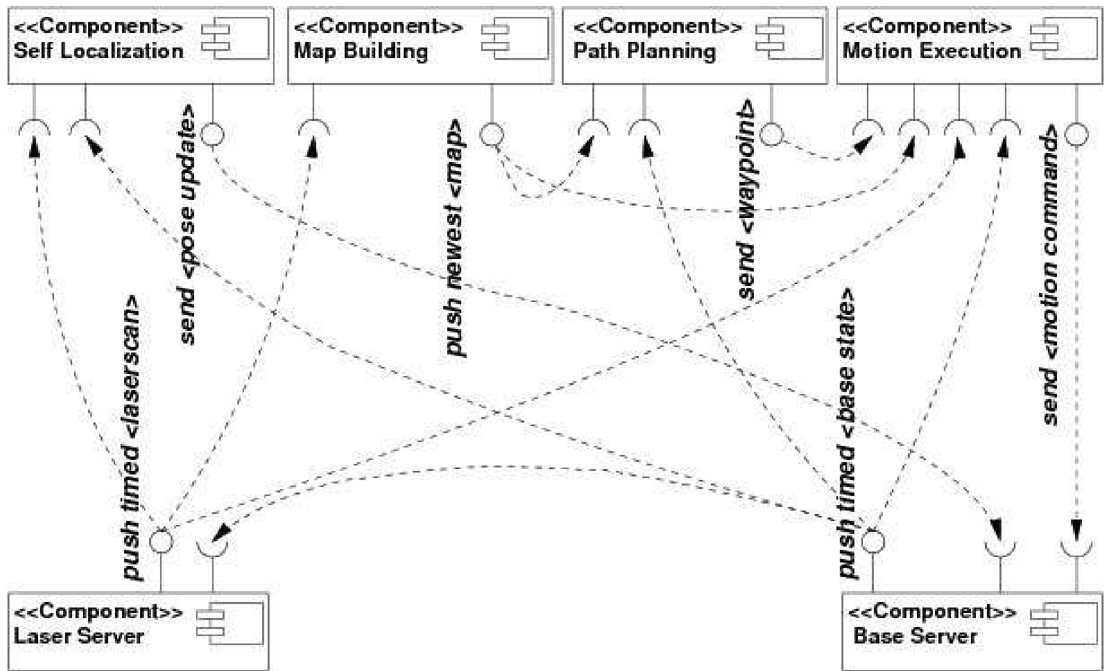


Figura 3.21: Ejemplo de Deployment para una aplicación de navegación

Los servicios que prestan y que requieren estos componentes se basan en los patrones de comunicación y objetos de comunicación estandarizados. Por lo tanto, uno puede fácilmente reemplazar los componentes, siempre y cuando los servicios prestados y requeridos necesarios para la aplicación estén disponibles de nuevo, y los puertos sean del mismo tipo. Como ejemplo, se pueden reutilizar los componentes de navegación en una plataforma diferente tan pronto como sus servicios requeridos y proporcionados tengan el equivalente correspondiente.

El cableado entre los componentes puede ser reorganizado en tiempo de ejecución si se utiliza el patrón Wiring, ya sea por los servicios propios del cliente desde dentro del componente o por otro componente del exterior. Esto proporciona máxima flexibilidad para la implementación del cableado dinámico como es necesario para la ejecución de tareas que dependan de la situación o que son sensibles al contexto.

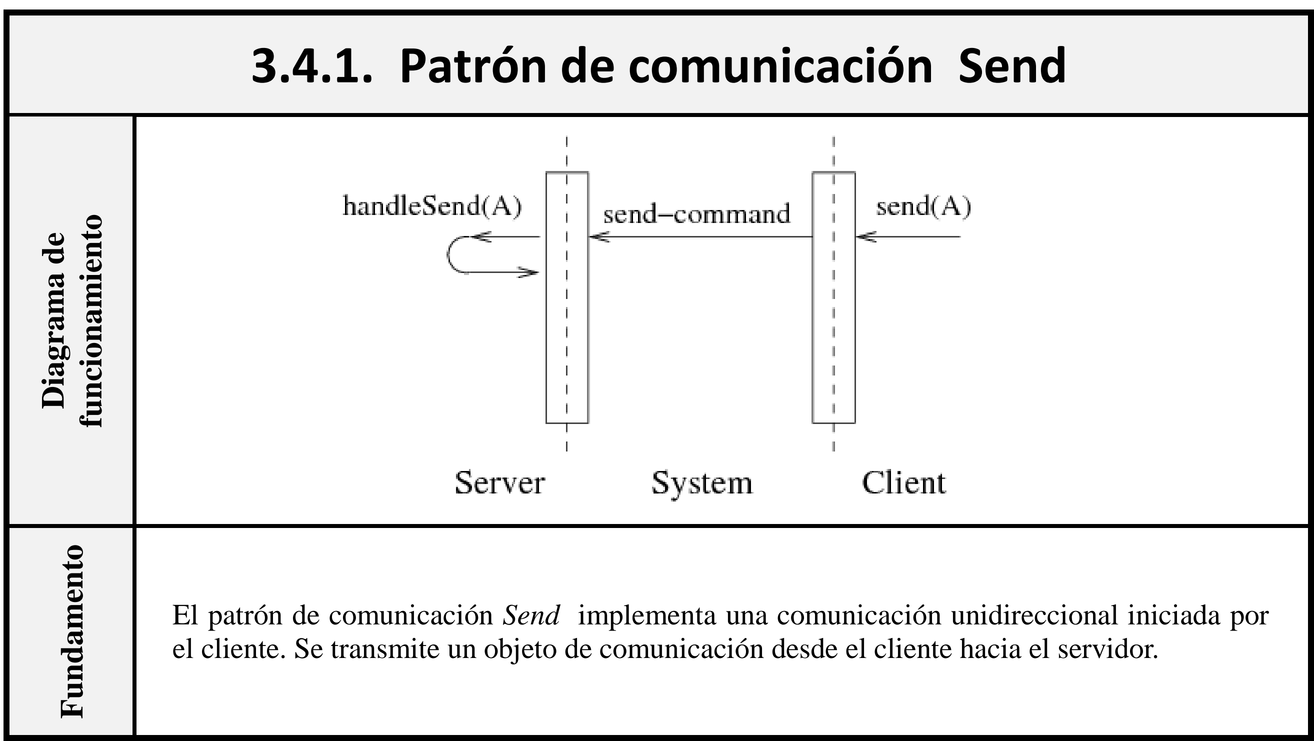
Puede haber incluso varias instancias del mismo servicio, como cuando varios componentes proporcionan funcionalidades similares, o incluso se superponen completamente. Debido a las conexiones explícitas, uno puede acceder directamente a implementaciones alternativas. Esto permite el balanceo de carga, así como seleccionar el planificador de ruta más adecuado para una situación específica entre un conjunto de planificadores de ruta alternativos; o modificar únicamente el componente *Base Server* si queremos ejecutar la aplicación en un robot distinto.

3.4. Patrones de Comunicación

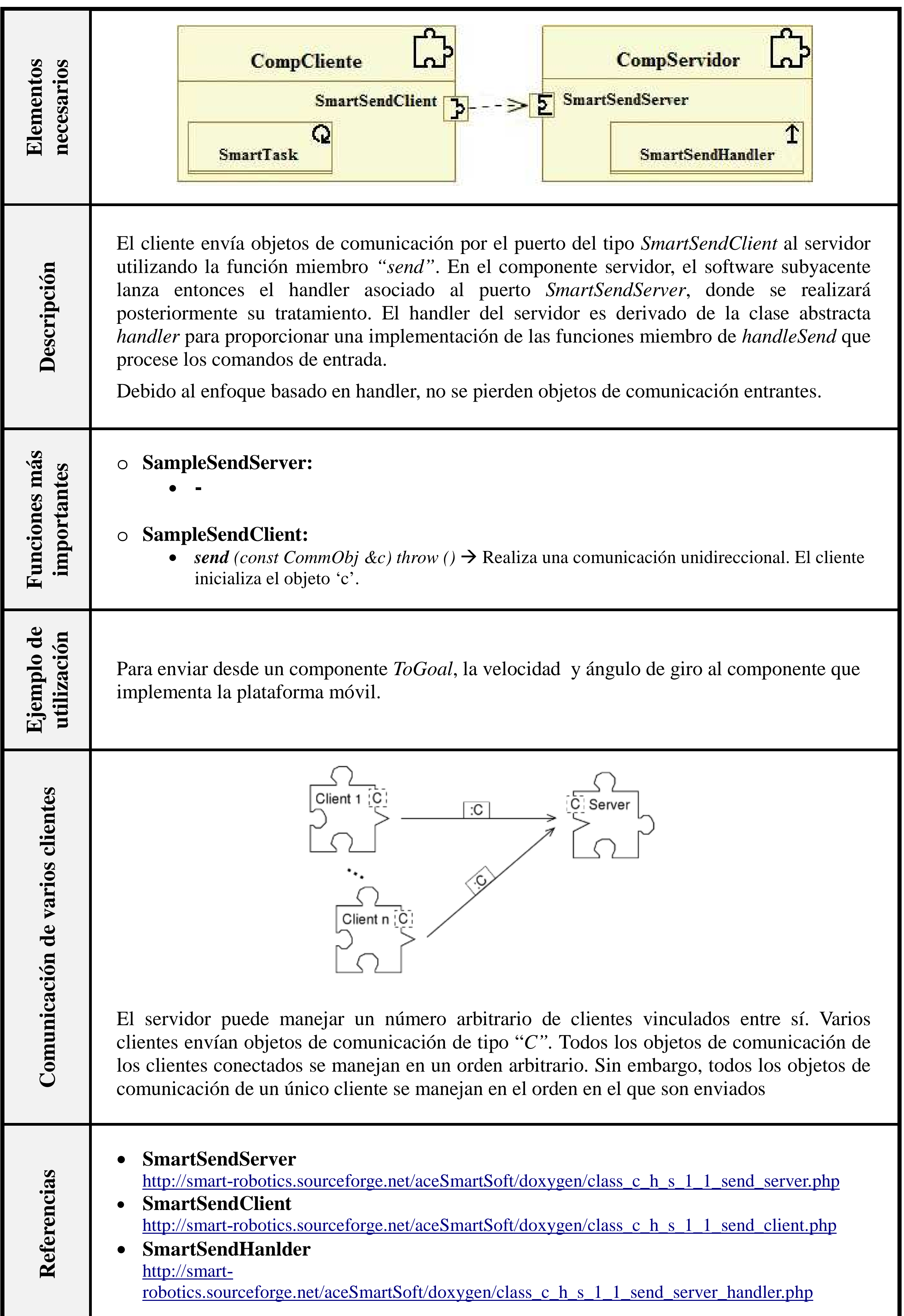
En este apartado se describe el conjunto de patrones de comunicación de SmartSoft, que ya anticipamos en el apartado *"3.2.3 – puertos de comunicación"*. El conjunto de patrones de comunicación es:

- *Patrón de comunicación Send*
- *Patrón de comunicación Query*
- *Patrón de comunicación PushNewest*
- *Patrón de comunicación PushTimed*
- *Patrón de comunicación Event*
- *Patrón de comunicación Wiring*
- *Patrón de comunicación State*

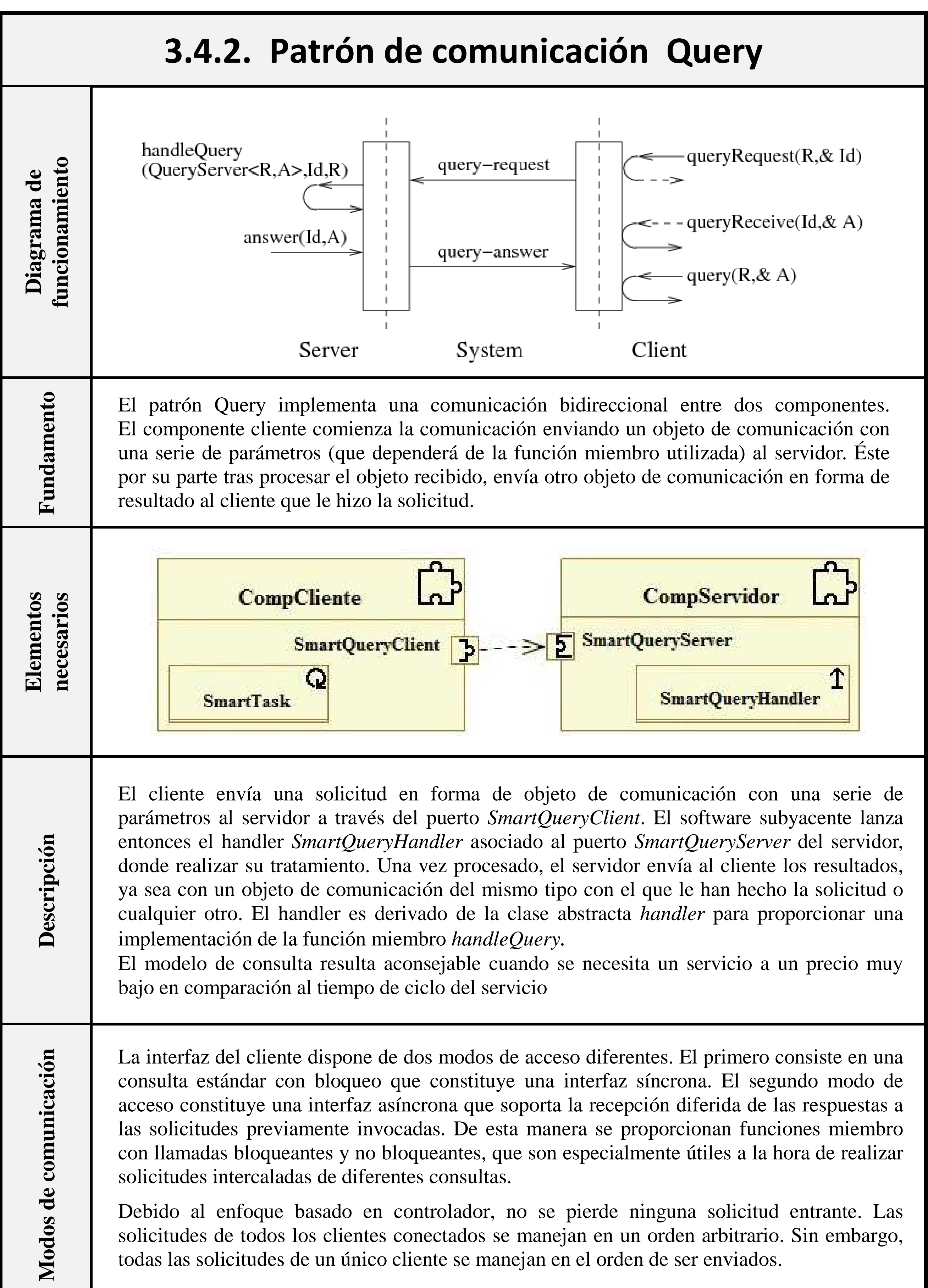
Sin embargo algunos de ellos se utilizan mucho más comúnmente que otros, y requieren por tanto de mayor atención. De cara a realizar una explicación clara y detallada de estos patrones más utilizados, se describen en forma de tabla.



3.4.1. Patrón de comunicación Send	
Diagrama de funcionamiento	
Fundamento	El patrón de comunicación *Send* implementa una comunicación unidireccional iniciada por el cliente. Se transmite un objeto de comunicación desde el cliente hacia el servidor.



Elementos necesarios	
Descripción	El cliente envía objetos de comunicación por el puerto del tipo *SmartSendClient* al servidor utilizando la función miembro *"send"*. En el componente servidor, el software subyacente lanza entonces el handler asociado al puerto *SmartSendServer*, donde se realizará posteriormente su tratamiento. El handler del servidor es derivado de la clase abstracta *handler* para proporcionar una implementación de las funciones miembro de *handleSend* que procese los comandos de entrada. Debido al enfoque basado en handler, no se pierden objetos de comunicación entrantes.
Funciones más importantes	o **SampleSendServer:** • **-** o **SampleSendClient:** • ***send*** *(const CommObj &c) throw ()* → Realiza una comunicación unidireccional. El cliente inicializa el objeto 'c'.
Ejemplo de utilización	Para enviar desde un componente *ToGoal*, la velocidad y ángulo de giro al componente que implementa la plataforma móvil.
Comunicación de varios clientes	El servidor puede manejar un número arbitrario de clientes vinculados entre sí. Varios clientes envían objetos de comunicación de tipo "*C*". Todos los objetos de comunicación de los clientes conectados se manejan en un orden arbitrario. Sin embargo, todos los objetos de comunicación de un único cliente se manejan en el orden en el que son enviados
Referencias	• **SmartSendServer** http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_send_server.php • **SmartSendClient** http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_send_client.php • **SmartSendHanlder** http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_send_server_handler.php



3.4.2. Patrón de comunicación Query

Diagrama de funcionamiento

Fundamento

El patrón Query implementa una comunicación bidireccional entre dos componentes. El componente cliente comienza la comunicación enviando un objeto de comunicación con una serie de parámetros (que dependerá de la función miembro utilizada) al servidor. Éste por su parte tras procesar el objeto recibido, envía otro objeto de comunicación en forma de resultado al cliente que le hizo la solicitud.

Elementos necesarios

Descripción

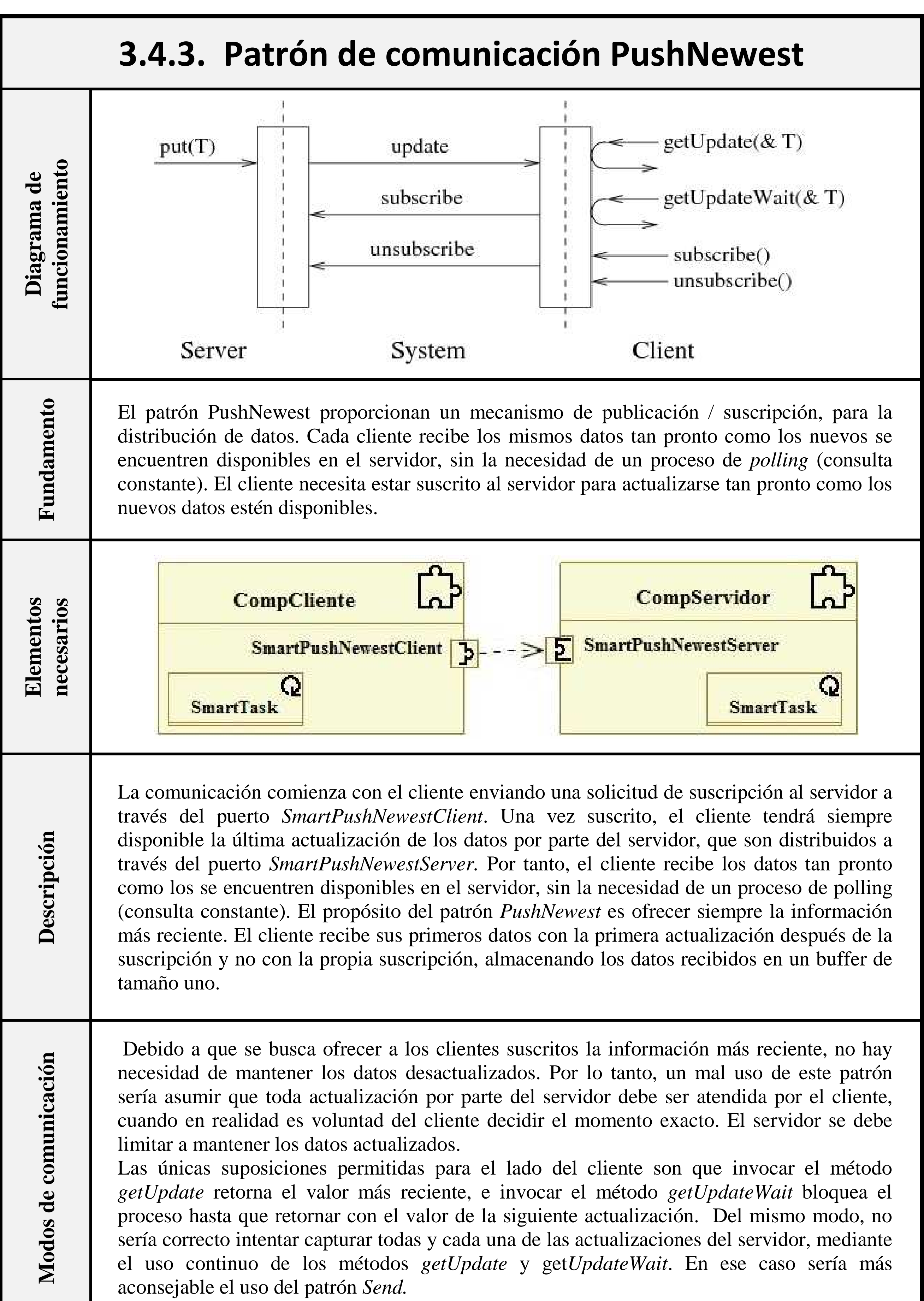
El cliente envía una solicitud en forma de objeto de comunicación con una serie de parámetros al servidor a través del puerto *SmartQueryClient*. El software subyacente lanza entonces el handler *SmartQueryHandler* asociado al puerto *SmartQueryServer* del servidor, donde realizar su tratamiento. Una vez procesado, el servidor envía al cliente los resultados, ya sea con un objeto de comunicación del mismo tipo con el que le han hecho la solicitud o cualquier otro. El handler es derivado de la clase abstracta *handler* para proporcionar una implementación de la función miembro *handleQuery*.
El modelo de consulta resulta aconsejable cuando se necesita un servicio a un precio muy bajo en comparación al tiempo de ciclo del servicio

Modos de comunicación

La interfaz del cliente dispone de dos modos de acceso diferentes. El primero consiste en una consulta estándar con bloqueo que constituye una interfaz síncrona. El segundo modo de acceso constituye una interfaz asíncrona que soporta la recepción diferida de las respuestas a las solicitudes previamente invocadas. De esta manera se proporcionan funciones miembro con llamadas bloqueantes y no bloqueantes, que son especialmente útiles a la hora de realizar solicitudes intercaladas de diferentes consultas.

Debido al enfoque basado en controlador, no se pierde ninguna solicitud entrante. Las solicitudes de todos los clientes conectados se manejan en un orden arbitrario. Sin embargo, todas las solicitudes de un único cliente se manejan en el orden de ser enviados.

Funciones más importantes	○ **SampleQueryServer:** • ***answer*** *(const QueryId id, const A &answer)* → Respuesta ante la petición del cliente. El servidor envía *'answer'* gracias al identificador *'id'* que identifica la solicitud a la que pertenece la respuesta. ○ **SampleQueryClient:** • ***query*** *(const R &request, A &answer)* → Consulta bloqueante. El cliente declara *'request'* y *'answer'* e inicializa *'request'*. El servidor inicializa *'answer'*. • ***queryRequest*** *(const R &request, QueryId &id)* → Consulta asíncrona. El cliente declara e inicializa *'id'* y *'request'*. El identificador *'id'* es utilizado por el servidor para enviar más tarde la respuesta. • ***queryReceive*** *(const QueryId id, A &answer)* → Comprueba si la respuesta está disponible. El cliente declara 'answer'. El servidor utiliza el identificador de consulta 'id' e inicializa el objeto *'answer'*. • ***queryReceiveWait*** *(const QueryId id, A &answer)* → Bloqueo en espera de la respuesta. El cliente declara 'answer'. El servidor utiliza el identificador de consulta 'id' e inicializa el objeto *'answer'*. • ***queryDiscard*** *(const QueryId id)* → Descarta la consulta con identificador 'id'.
Ejemplo de utilización	Para solicitar una determinada parte de un mapa, donde en el objeto de comunicación que hace las veces de petición se especifica el tamaño y el punto de origen del mismo. Más tarde, el servidor o proveedor del servicio devuelve la respuesta en el objeto de comunicación que corresponda.
Comunicación de varios clientes	El servidor puede manejar un número arbitrario de clientes. Las solicitudes de todos los clientes conectados se manejan en un orden arbitrario. Sin embargo, todas las solicitudes de un único cliente se manejan en el orden de ser enviados.
Referencias	• **SmartQueryServer** http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_query_server.php • **SmartQueryClient** http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_query_client.php • **SmartQueryHanlder** http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_query_server_handler.php



3.4.3. Patrón de comunicación PushNewest

Diagrama de funcionamiento

Fundamento

El patrón PushNewest proporcionan un mecanismo de publicación / suscripción, para la distribución de datos. Cada cliente recibe los mismos datos tan pronto como los nuevos se encuentren disponibles en el servidor, sin la necesidad de un proceso de *polling* (consulta constante). El cliente necesita estar suscrito al servidor para actualizarse tan pronto como los nuevos datos estén disponibles.

Elementos necesarios

Descripción

La comunicación comienza con el cliente enviando una solicitud de suscripción al servidor a través del puerto *SmartPushNewestClient*. Una vez suscrito, el cliente tendrá siempre disponible la última actualización de los datos por parte del servidor, que son distribuidos a través del puerto *SmartPushNewestServer*. Por tanto, el cliente recibe los datos tan pronto como los se encuentren disponibles en el servidor, sin la necesidad de un proceso de polling (consulta constante). El propósito del patrón *PushNewest* es ofrecer siempre la información más reciente. El cliente recibe sus primeros datos con la primera actualización después de la suscripción y no con la propia suscripción, almacenando los datos recibidos en un buffer de tamaño uno.

Modos de comunicación

Debido a que se busca ofrecer a los clientes suscritos la información más reciente, no hay necesidad de mantener los datos desactualizados. Por lo tanto, un mal uso de este patrón sería asumir que toda actualización por parte del servidor debe ser atendida por el cliente, cuando en realidad es voluntad del cliente decidir el momento exacto. El servidor se debe limitar a mantener los datos actualizados.

Las únicas suposiciones permitidas para el lado del cliente son que invocar el método *getUpdate* retorna el valor más reciente, e invocar el método *getUpdateWait* bloquea el proceso hasta que retornar con el valor de la siguiente actualización. Del mismo modo, no sería correcto intentar capturar todas y cada una de las actualizaciones del servidor, mediante el uso continuo de los métodos *getUpdate* y get*UpdateWait*. En ese caso sería más aconsejable el uso del patrón *Send.*

Funciones más importantes

- **SamplePushTimedServer:**
  - ***put*** *(const T &d)* → Envía los datos actualizados a todos los clients suscritos. El objeto 'T' es declarado e inicializado en el servidor.

- **SamplePushTimedClient:**
  - ***subscribe*** *()* → Subscribe el cliente al servidor, para obtener las actualizaciones tan pronto como el segundo las tenga disponibles.
  - ***unsubscribe*** *()* → Elimina la suscripción al servidor, con lo que no recibirá más actualizaciones.
  - ***blocking*** *(const bool b)* → Permite o rechaza las llamadas bloqueantes.
  - ***getUpdate*** *(T &d)* → Llamada no bloqueante que retorna de manera inmediata el valor del buffer de datos del cliente, que contiene la actualización más reciente distribuida por el servidor. El objeto *'T'* es declarado en el cliente.
  - ***getUpdateWait*** *(T &d)* → Llamada bloqueante que espera hasta la siguientes actualización distribuida por el servidor. El objeto *'T'* es declarado en el cliente.

Ejemplo de utilización

Este patrón resulta conveniente cuando varios clientes necesiten los mismos datos, o en el caso de que el régimen de actualización queremos que sea dictado por el servidor. Los patrones push se utilizan, por ejemplo, para distribuir los datos leídos por el láser a diferentes componentes y para proporcionar una revisión actualizada del mapa local tan pronto como algo ha cambiado (que es conocido en el lado del servidor).

Comunicación de varios clientes

La comunicación es iniciada por el cliente suscribiéndose al servidor. Sin embargo ese envío no contiene objeto de comunicación alguno. A partir de ahí el servidor distribuye los objetos de comunicación *'D'* a los clientes, pudiendo manejar un número arbitrario de éstos.

Referencias

- **SmartPushNewestServer**
  http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_push_newest_server.php
- **SmartPushNewestClient**
  http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_push_newest_client.php

3.4.4. Patrón de comunicación PushTimed

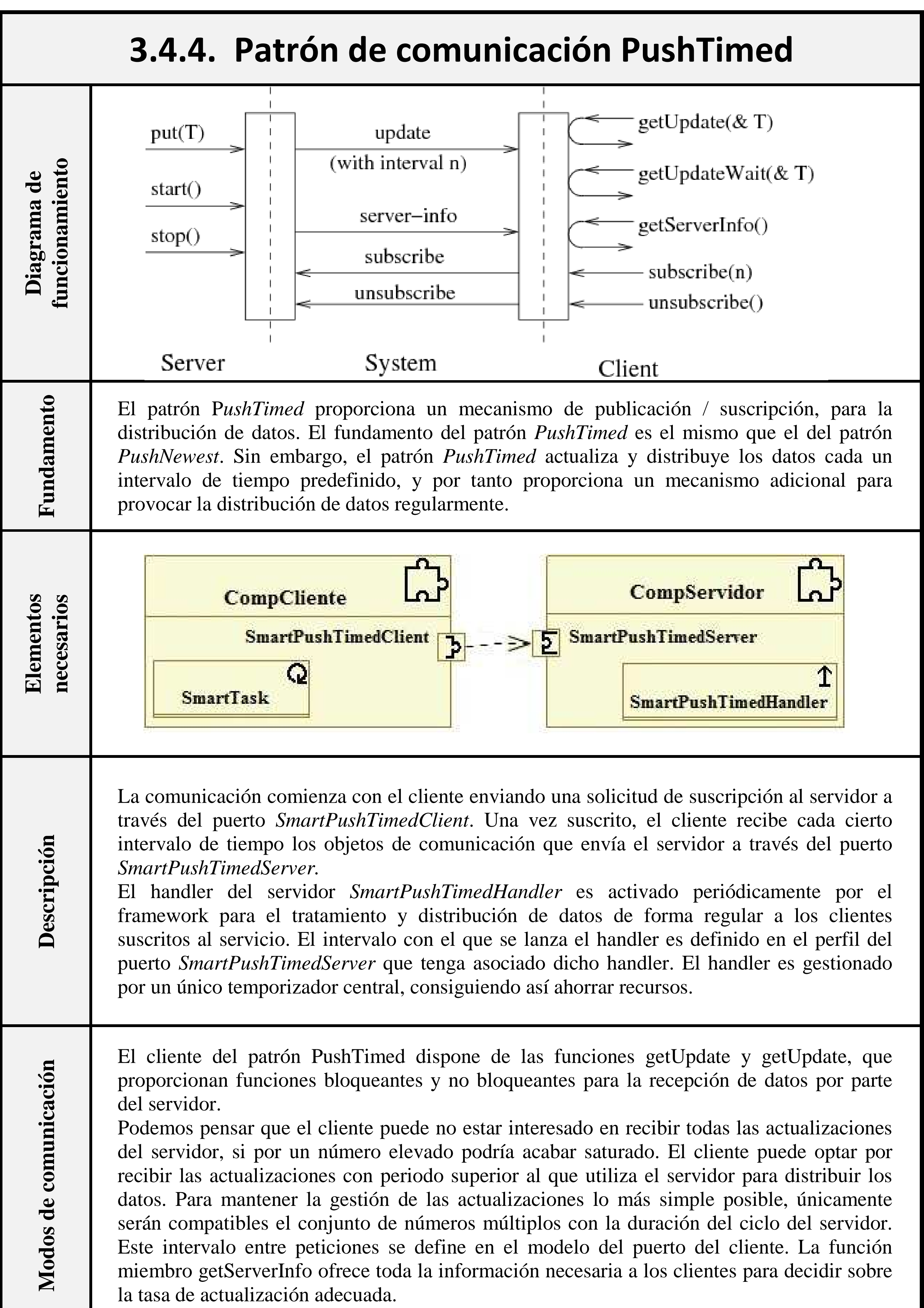


Diagrama de funcionamiento

Fundamento

El patrón P*ushTimed* proporciona un mecanismo de publicación / suscripción, para la distribución de datos. El fundamento del patrón *PushTimed* es el mismo que el del patrón *PushNewest*. Sin embargo, el patrón *PushTimed* actualiza y distribuye los datos cada un intervalo de tiempo predefinido, y por tanto proporciona un mecanismo adicional para provocar la distribución de datos regularmente.

Elementos necesarios

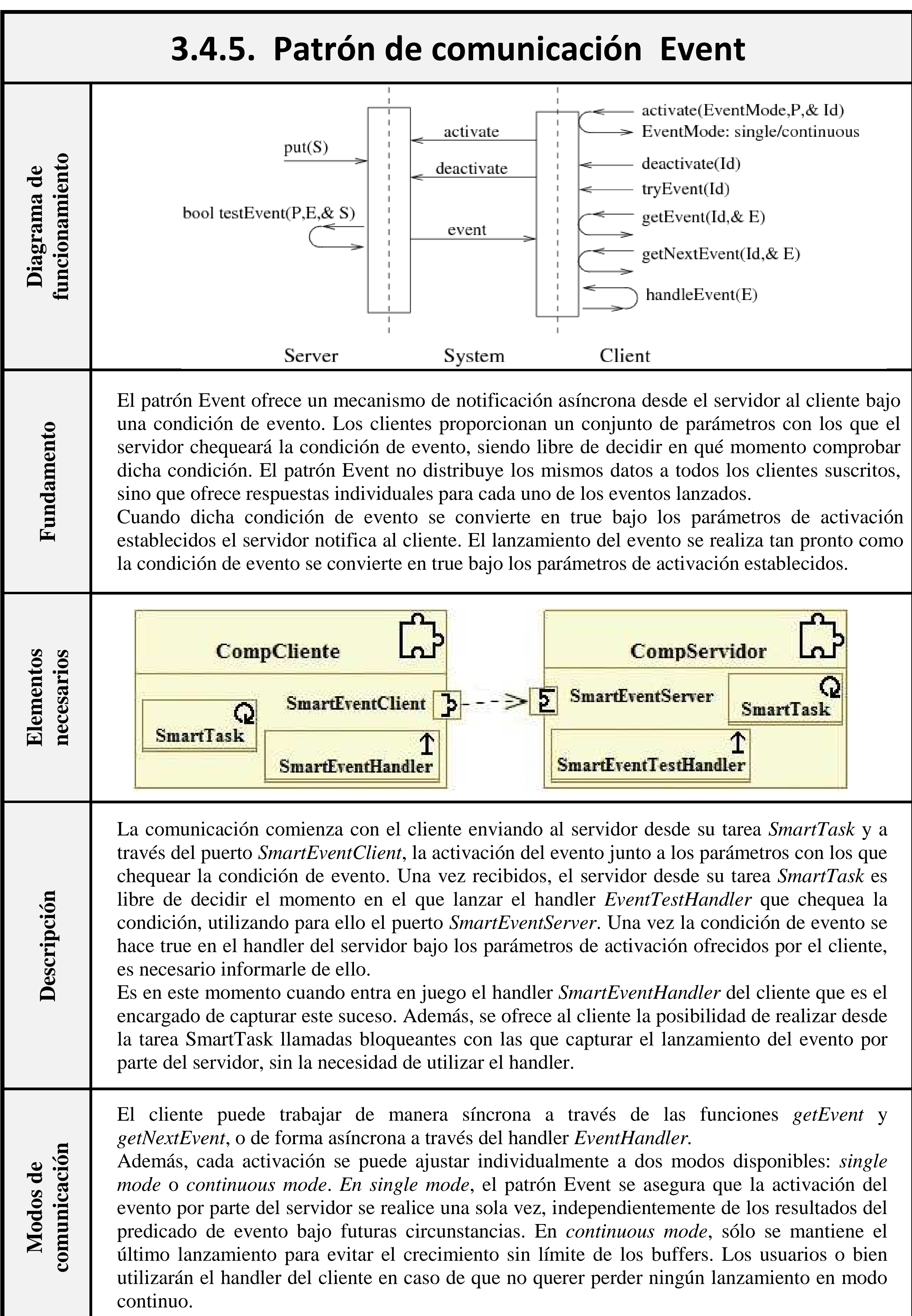
Descripción

La comunicación comienza con el cliente enviando una solicitud de suscripción al servidor a través del puerto *SmartPushTimedClient*. Una vez suscrito, el cliente recibe cada cierto intervalo de tiempo los objetos de comunicación que envía el servidor a través del puerto *SmartPushTimedServer*.
El handler del servidor *SmartPushTimedHandler* es activado periódicamente por el framework para el tratamiento y distribución de datos de forma regular a los clientes suscritos al servicio. El intervalo con el que se lanza el handler es definido en el perfil del puerto *SmartPushTimedServer* que tenga asociado dicho handler. El handler es gestionado por un único temporizador central, consiguiendo así ahorrar recursos.

Modos de comunicación

El cliente del patrón PushTimed dispone de las funciones getUpdate y getUpdate, que proporcionan funciones bloqueantes y no bloqueantes para la recepción de datos por parte del servidor.
Podemos pensar que el cliente puede no estar interesado en recibir todas las actualizaciones del servidor, si por un número elevado podría acabar saturado. El cliente puede optar por recibir las actualizaciones con periodo superior al que utiliza el servidor para distribuir los datos. Para mantener la gestión de las actualizaciones lo más simple posible, únicamente serán compatibles el conjunto de números múltiplos con la duración del ciclo del servidor. Este intervalo entre peticiones se define en el modelo del puerto del cliente. La función miembro getServerInfo ofrece toda la información necesaria a los clientes para decidir sobre la tasa de actualización adecuada.

Funciones más importantes	○ **SamplePushTimedServer:** • ***start*** *()* → Activa el trigger con el que se lanzará el handler del servidor. • ***stop*** *()* → Desactiva el trigger utilizado por el handler del servidor. • ***put*** *(const T &d)* → El servidor envía el objeto 'T' a todos los clientes suscritos, teniendo en cuenta los intervalos de actualización individuales que éstos posean. ○ **SamplePushTimedClient:** • ***subscribe*** *(const int interval)* → Suscribe al cliente para recibir notificaciones con un interval *'interval'*. • ***unsubscribe*** *()* → Elimina la suscripción al cliente de modo que no reciba más actualizaciones. • ***blocking*** *(const bool b)* → Permite o rechaza las llamadas bloqueantes. • ***getUpdate*** *(T &d)* → Llamada no bloqueante que retorna de manera inmediata la actualización más reciente distribuida por el servidor. El objeto *'T'* es declarado en el cliente. • ***getUpdateWait*** *(T &d)* → Llamada bloqueante que espera hasta la siguientes actualización distribuida por del servidor. El objeto *'T'* es declarado en el cliente. • ***getServerInfo*** *(double &t, bool &r)* → Get cycle time and server state.
Ejemplo de utilización	En un componente de acceso al hardware específico, que proporcionan acceso a un láser. Este componente hace las veces de servidor, y envía periódicamente las lecturas obtenidas a todos los componentes clientes suscritos que las requieran.
Comunicación de varios clientes	La comunicación es iniciada por el cliente suscribiéndose al servidor. Sin embargo ese envío no contiene objeto de comunicación alguno. A partir de ahí el servidor distribuye los objetos de comunicación *'D'* a los clientes, pudiendo manejar un número arbitrario de éstos.
Referencias	• **SmartPushTimedServer** http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_push_timed_server.php • **SmartPushTimedClient** http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_push_timed_client.php • **SmartPushTimedHanlder** http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_push_timed_handler.php



3.4.5. Patrón de comunicación Event

Diagrama de funcionamiento

Fundamento

El patrón Event ofrece un mecanismo de notificación asíncrona desde el servidor al cliente bajo una condición de evento. Los clientes proporcionan un conjunto de parámetros con los que el servidor chequeará la condición de evento, siendo libre de decidir en qué momento comprobar dicha condición. El patrón Event no distribuye los mismos datos a todos los clientes suscritos, sino que ofrece respuestas individuales para cada uno de los eventos lanzados.
Cuando dicha condición de evento se convierte en true bajo los parámetros de activación establecidos el servidor notifica al cliente. El lanzamiento del evento se realiza tan pronto como la condición de evento se convierte en true bajo los parámetros de activación establecidos.

Elementos necesarios

Descripción

La comunicación comienza con el cliente enviando al servidor desde su tarea *SmartTask* y a través del puerto *SmartEventClient*, la activación del evento junto a los parámetros con los que chequear la condición de evento. Una vez recibidos, el servidor desde su tarea *SmartTask* es libre de decidir el momento en el que lanzar el handler *EventTestHandler* que chequea la condición, utilizando para ello el puerto *SmartEventServer*. Una vez la condición de evento se hace true en el handler del servidor bajo los parámetros de activación ofrecidos por el cliente, es necesario informarle de ello.
Es en este momento cuando entra en juego el handler *SmartEventHandler* del cliente que es el encargado de capturar este suceso. Además, se ofrece al cliente la posibilidad de realizar desde la tarea SmartTask llamadas bloqueantes con las que capturar el lanzamiento del evento por parte del servidor, sin la necesidad de utilizar el handler.

Modos de comunicación

El cliente puede trabajar de manera síncrona a través de las funciones *getEvent* y *getNextEvent*, o de forma asíncrona a través del handler *EventHandler*.
Además, cada activación se puede ajustar individualmente a dos modos disponibles: *single mode* o *continuous mode*. *En single mode*, el patrón Event se asegura que la activación del evento por parte del servidor se realice una sola vez, independientemente de los resultados del predicado de evento bajo futuras circunstancias. En *continuous mode*, sólo se mantiene el último lanzamiento para evitar el crecimiento sin límite de los buffers. Los usuarios o bien utilizarán el handler del cliente en caso de que no querer perder ningún lanzamiento en modo continuo.

Funciones más importantes

- **SampleEventServer:**
  - ***put*** *(const S &state)* → Inicia el chequeo de la condición de evento. ‘s’ es la información actual frente a los que chequear los parámetros de activación recibidos.

- **SampleEventTestHandler:**
  - ***testEvent*** *(P &p, E &e, const S &s)* → Método de test que se encarga de decidir si se lanza el evento o no. ‘p’ son los parámetros de activación a chequear, ‘e’ es el objeto de respuesta al evento, y ‘s’ es la información actual frente a la que chequear los parámetros.

- **SmartEventClient:**
  - ***activate*** *(const EventMode mode, const P ¶meter, EventId &id)* → Activa un evento proporcionando el modo *(‘single’ o ‘continuous’),* los parámetros de activación *‘p’* y el identificador de evento *‘id’*.
  - ***deactivate*** *(const EventId id)* → Desactiva el evento especificado con identificador *‘id’*.
  - ***tryEvent*** *(const EventId id)* → Comprueba si el evento con identificador ‘id’ ha sido lanzado.
  - ***getEvent*** *(const EventId id, E &event)* → Llamada bloqueante que espera hasta que el evento es lanzado por parte del cliente y tras lo cual lo finaliza (tiene en cuenta estado actual). El objeto *‘event’* es declarado en el cliente e inicializado en el servidor.
  - ***getNextEvent*** *(const EventId id, E &event)* → Llamada bloqueante que espera la próxima activación del evento (no tienen en cuenta el estado actual). El objeto *‘event’* es declarado en el cliente e inicializado en el servidor.

- **SmartEventHandler:**
  - ***handleEvent*** *(const EventId id, const E &event)* → Handler invocado por el patrón de comunicación del cliente cada vez que el evento es lanzado desde el servidor. El parámetro *‘id’* es el identificador de evento, y el *‘event’* el objeto declarado en el cliente e inicializado en el servidor.

Ejemplo de utilización

Este patrón es utilizado para sincronizar de manera continua el progreso en la ejecución de las tareas. Un ejemplo de utilización lo tenemos en el control de la tensión de la batería del robot. Una activación podría comprobar las caídas de tensión por debajo de un umbral de alerta, otra activación del mismo evento podría comprobar si el nivel de tensión está por debajo de un umbral crítico, y otra activación puede ser parametrizada de tal manera que se lance cuando la tensión vuelva a estar en el rango esperado.

Comunicación de varios clientes

Un servidor de eventos puede soportar la activación de distintos eventos incluso de diferentes clientes. Cada uno tendrá sus propios parámetros ‘P’ para el predicado del evento. Cada activación es analizada de forma individual y lanzada de acuerdo a sus parámetros de activación individual.

Referencias

- **SmartEventServer**
  http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_event_server.php
- **SmartEventTestHandler**
  http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_event_test_handler.php
- **SmartEventClient**
  http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_event_client.php
- **SmartEventHandler**
  http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_event_handler.php

3.4.6. Patrón de comunicación “Wiring”

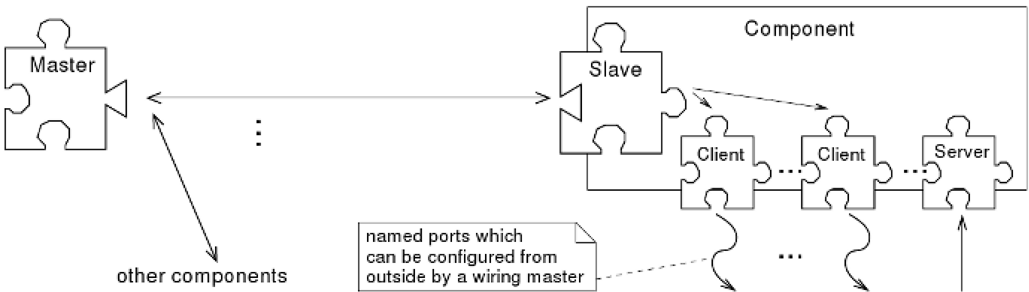


Figura 3.22: Patrón Wiring

El patrón Wiring proporciona un mecanismo coherente para el cableado dinámico de las partes cliente (solicitantes de servicio) de los patrones de comunicación desde fuera de un componente. El puerto que solicita el servicio se puede exponer a sí mismo como wireable desde fuera del componente, para inscribirse en el *Wiring Slave* del componente. Se convierte así en un cliente administrado, pudiendo entonces ser conectado a un proveedor de servicio apropiado al margen del componente a través de la parte maestra del patrón Wiring (*Wiring Master*).

Un solicitante de servicio puede conectarse a cualquier proveedor de servicio siempre y cuando ambos sean compatibles en términos del servicio. La compatibilidad en cuanto a servicios requiere que ambas partes pertenezcan al mismo tipo de patrón de comunicación y que ambas partes sean parametrizadas por los mismos tipos de objetos de comunicación.

El cableado en tiempo de ejecución permite realizar en cualquier momento todo tipo de cambios en las conexiones entre las partes cliente y servidor de un servicio, y requiere de los patrones de comunicación una implementación que permita hacer frente adecuadamente a estos cambios de conexión, incluso cuando están pendientes de actividades de comunicación. Todo ello, se maneja desde dentro de los patrones de comunicación y de manera oculta al usuario, lo cual reduce enormemente la complejidad que este advierte.

El cableado dinámico es necesario por ejemplo, para cambiar el flujo de datos entre componentes que componen los diferentes comportamientos de un conjunto de habilidades.

El patrón Wiring difiere del resto de patrones de comunicación en que no requiere de objetos de comunicación en su funcionamiento. Por otra parte, el *Wiring Slave* no posee ninguna función miembro que pueda ser utilizada por el usuario, realizando todo su trabajo de forma transparente y sin necesidad de que éste realice interacción alguna.

La funciones miembro *connect (…)* y *disconnect (…)* del *Wiring Master*, no deben ser confundidas con las funciones miembro que poseen los solicitantes de servicios de otros patrones de comunicación. La función miembro *connect (…)* no establece una conexión desde el *Wiring Master* al *Wiring Slave*, sino que conecta un cliente o solicitante de servicio administrado, con un proveedor de servicio acorde. Un *Wiring Master* puede conectar arbitrariamente clientes de componentes gestionados, sin ser explícitamente conectado a un *Wiring Slave* antes de emitir una orden de cableado. La integración del patrón Wiring en un componente se ilustra en la figura 3.22.

Aplicar el patrón Wiring sólo requiere crear una instancia de las clases de cableado. Un componente puede tener un único *Wiring Slave*. Los solicitantes de servicios pueden registrarse en el *Wiring Slave* del componente haciendo uso de las funciones miembro *add (…)* y *remove (…).* Independientemente de ser un cliente “administrado” o no, aún se pueden utilizar las funciones miembro *connect (…)* y *disconnect (…)* de los solicitantes de servicios para establecer el cableado entre componentes. El cableado que finalmente se hará efectivo está determinado exclusivamente por el orden de las llamadas. Un *Wiring Master* también puede ser utilizado para configurar los clientes administrados de su propio componente.

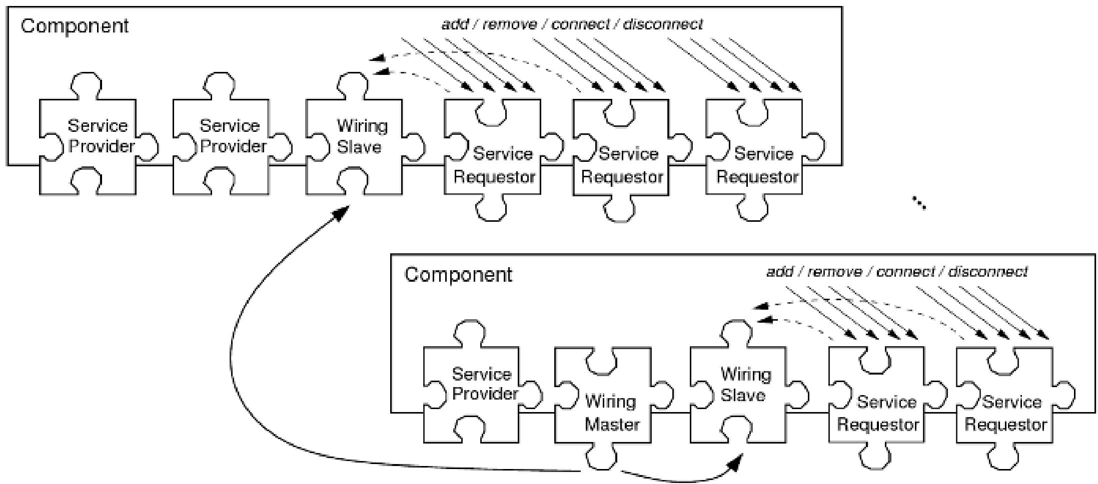


Figura 3.23: Wiring slaves

Encontramos más información sobre este patrón de comunicación en las siguientes direcciones web:

- ***WiringMaster***
  http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_wiring_master.php
- ***WiringSlave***
  http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_wiring_slave.php

3.4.7. Patrón de comunicación "State"

El patrón de comunicación State proporciona una relación maestro-esclavo para activar y desactivar selectivamente estados. Una actividad puede bloquear un estado en el esclavo para inhibir los cambios de estado en las secciones críticas, como se muestra en la figura 3.22.

Una sección crítica impide que una actividad pueda ser interrumpida en un punto inadecuado de su ejecución. El patrón State da prioridad al maestro para efectuar cambios de estado sobre el esclavo.

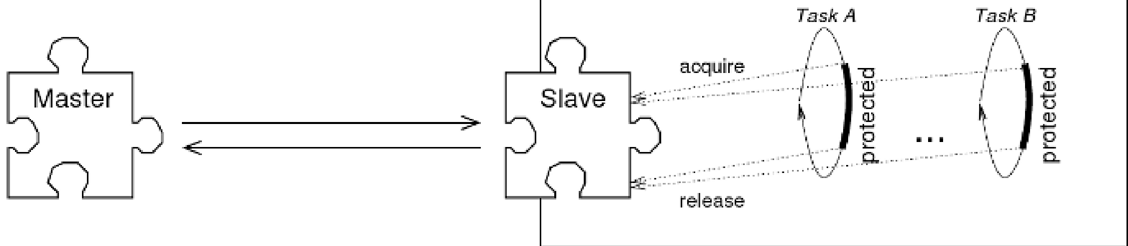


Figura 3.24: State Master/Slave

Tan pronto como se recibe la solicitud de cambio de estado del maestro, el esclavo rechaza los bloqueos para los estados que no son compatibles con el cambio de estado en espera del maestro. La solicitud de cambio de estado que realizó el maestro se ejecuta por el esclavo tan pronto como todos los bloqueos de los estados afectados por el cambio de estado son liberados.

El patrón State se utiliza por ejemplo para la supervisión de las tareas de un componente en la secuencia de las distintas capas a la hora de la correcta desactivación de las actividades internas de usuario de los componentes.

El patrón State se utiliza para supervisar las tareas de un componente, por ejemplo a la hora de realizar correctamente la secuencia de desactivación de las distintas capas de actividades internas de usuario de los componentes. Se proporcionan en el esclavo handlers que permiten las tareas de limpieza con cambio de estado. Información adicional sobre este patrón se puede encontrar en la tesis doctoral del Prof. Dr. C. Schlegel (apartado 6.2).

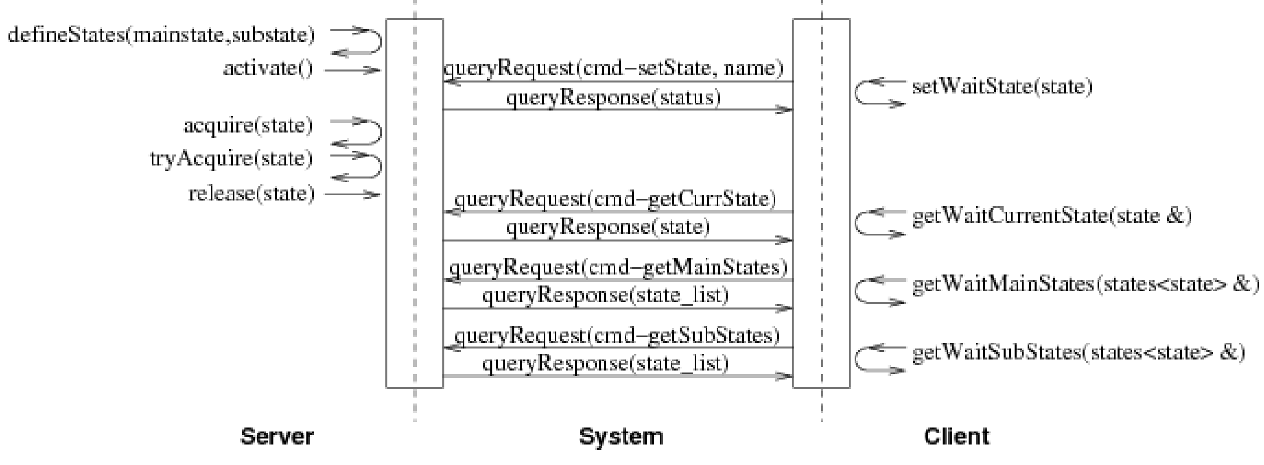


Figura 3.25: Patrón State

Encontramos más información sobre este patrón de comunicación en las siguientes direcciones web:

- ***StateMaster***
  http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_state_master.php
- ***StateSlave***
  http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/class_c_h_s_1_1_state_slave.php

4. PRIMEROS PASOS CON SMARTSOFT

4.1. Puesta en marcha SmartSoft

Para la puesta en marcha de SmartSoft en un equipo, se ofrecen dos posibilidades: el uso de una imagen VMWare proporcionada por el equipo de desarrollo de SmartSoft, o seguir el tradicional proceso de instalación.

VMWare es un sistema de virtualización por software. Una máquina virtual es un software que emula a una computadora y puede ejecutar programas como si fuese una computadora real. Cuando se ejecuta el programa (simulador), proporciona un ambiente de ejecución similar a todos los efectos a un computador físico (excepto en el puro acceso físico al hardware simulado), con CPU (puede ser más de una), BIOS, tarjeta gráfica, memoria RAM, tarjeta de red, sistema de sonido, conexión USB, disco duro (pueden ser más de uno), etc. Un virtualizador por software permite ejecutar (simular) varios computadores (sistemas operativos) dentro de un mismo hardware de manera simultánea, permitiendo así el mayor aprovechamiento de recursos. No obstante, y al ser una capa intermedia entre el sistema físico y el sistema operativo que funciona en el hardware emulado, la velocidad de ejecución de este último es menor, pero en la mayoría de los casos suficiente para usarse en entornos de producción.

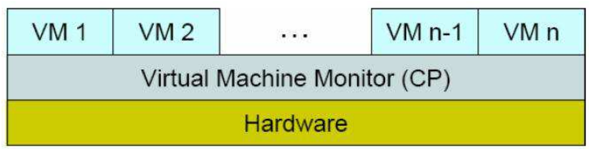


Figura 4.1: Arquitectura típica de una máquina virtual.

Mientras que la instalación SVN (controlador de versiones) de SmartSoft requiere una elevada cantidad de tiempo, haciendo uso de la máquina virtual, se realiza un esfuerzo bastante bajo a la hora de la puesta en marcha del sistema para empezar a trabajar con SmartSoft, ya que se ofrece una instalación de VMWare con todos los paquetes necesarios. Se proporciona así, una manera rápida y sencilla de familiarizarse con SmartSoft, no dependiendo en ningún momento de un modelo concreto de robot real. Esto se logra mediante el uso del simulador integrado proporcionado en la imagen (basado en *Player Stage*).

Si trabajamos con la imagen VMWare, no hay necesidad de modificar la configuración actual de su equipo, así como la posibilidad de utilizar la imagen de VMWare en nuestro sistema operativo favorito. También logramos una cierta tranquilidad por la despreocupación que supone el poder acabar estropeando algo llegando a una configuración errónea, y que siempre sea posible volver a nuestra imagen original de VMWare.

Los pasos a seguir para la puesta en marcha de SmartSoft haciendo uso de la imagen VMWare, los encontramos en el anexo A. Por otro lado, se describe el proceso de instalación completo en el anexo B.

4.2. Descripción de la herramienta de desarrollo

Tras completar el proceso de instalación, para comenzar a trabajar con SmartSoft debemos hacer doble clic en el icono *"SmartSoft MDSD toolchain"* de la interfaz gráfica de usuario (GUI) de Linux Ubuntu. A continuación se abrirá una ventana de diálogo donde seleccionar el espacio de trabajo o workspace en el que queremos trabajar. Tanto si hemos realizado el proceso de instalación como si hacemos uso de la imagen virtual, se incluyen de partida 3 espacios de trabajo distintos. Dependiendo del workspace en el que iniciemos la aplicación aparecerán a modo de ejemplo una serie de proyectos u otros. A continuación se detalla que tipo de proyectos encontramos en cada uno de ellos.

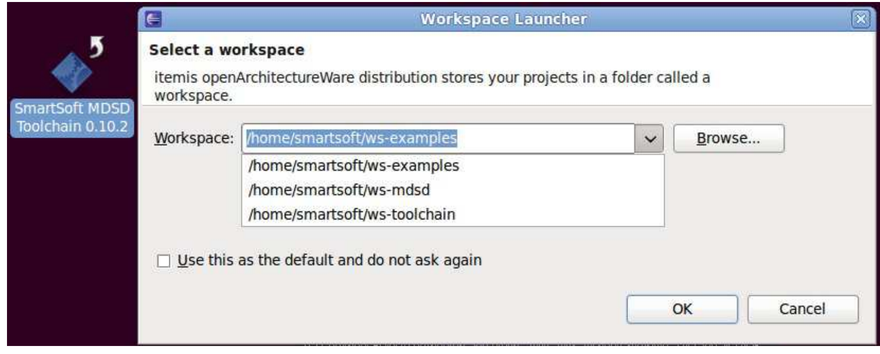


Figura 4.2: Selección del espacio de trabajo

ws-examples

Este workspace contiene una serie de proyectos donde se muestran mediante sencillos ejemplos, el uso y funcionamiento de los patrones de comunicación que proporciona SmartSoft.

En este workspace aparece un proyecto denominado *CommSampleObjects*, donde encontramos una colección de objetos de comunicación básicos. Alguno de los objetos que encontramos son: *CommSampleValues*, que no es más que un entero de 32 bits; *CommSamplePrint*, que simplemente es una cadena de texto; o *CommSampleTime,* que consta de un registro de enteros sin signo cuyos campos son "*hora, minuto y segundo*".

Aparecen también una serie de proyectos en los que están implementados componentes que se corresponden a los roles de clientes y servidor para cada uno de los patrones de comunicación: *SmartSendClient, SmartSendServer, SmartQueryClient, SmartQueryServer, SmartPushNewestClient, SmartPushNewestServer...*

Por último, tenemos un conjunto de proyectos donde encontramos los deployments asociados a cada patrón de comunicación, en los que se hace uso y se muestra el funcionamiento de los componentes antes mencionados: *DeploySampleSend, DeploySampleQuery, DeploSamplePushNewest...*

Este workspace será el idóneo para una primera toma de contacto con SmartSoft, ya que proporciona una manera rápida de entender el funcionamiento de los patrones de comunicación.

ws-mdsd

Este workspace contiene una serie de proyectos que serán de gran utilidad a la hora de realizar una aplicación robótica. Algunos de los componentes que encontraremos son:

- *SmartLaserObstacleAvoid,* que implementa un algoritmo para la evitación de obstáculos reactiva.
- *SmartLaserServer,* componente que modela el SICK LMS200 laser ranger.
- *SmartPioonerBaseServer,* componente que modela el robot Pioneer P3DX.
- *SmartPlayerStageSimulator,* es el componente del simulador *Player Stage*.

También son de gran importancia los objetos de comunicación que se proporcionan, que se encuentran clasificados en distintos proyectos según sea la naturaleza de la aplicación: *CommBasicObjects, CommNavigationObjects, CommManipulatorObjects...* Como pasaba en el workspace anterior, en este también aparecen deployments que hacen uso de los componentes antes nombrados. Ejemplos de estos son:

- *DeployLaserObstacleAvoidRobot,* implementación de una aplicación de evitación de obstáculos reactiva haciendo uso del robot Pioneer P3DX.
- *DeployLaserObstacleAvoid*, semejante al anterior pero haciendo uso del componente simulador en lugar del que modela al robot real.
- *DeployNavTask*, aplicación de planificación de trayectorias o path planning con el robot Pioneer P3DX.

Al contrario que pasaba en el workspaces anterior donde los componentes estaban implementados para usarlos en un único deployment, en este workspace advertimos como un componente es instanciado en varios deployment (el componente que modela el robot será el mismo para el *DeployLaserObstacleAvoid* que para el *DeployNavTask*). Este hecho refuerza la idea de que un componente, si está bien diseñado, puede ser reutilizado en múltiples contextos.

Este workspace requiere de un mayor conocimiento de SmartSoft que el anterior, pero será el más indicado cuando el usuario desee construir una aplicación robótica de utilidad, debido a la posibilidad de hacer uso de alguno de estos componentes proporcionados.

ws-toolchain

Este espacio de trabajo contiene los proyectos fuente de los plugins de MDSD de SmartSoft.Una simple cadena de herramientas de desarrollo de software consiste de un editor de texto para editar código fuente, un compilador y enlazador para transformar el código fuente en un programa ejecutable, bibliotecas para proveer una interfaz al sistema operativo, y un depurador.

Este workspace es utilizado por los desarrolladores del MDSD de SmartSoft, y por tanto es un entorno de trabajo donde los proyectos que se ofrecen, van más allá de los objetivos de nuestro proyecto.

Como conclusión, podemos decir que en cierta manera existe una analogía entre los espacios de trabajo proporcionados, y el tipo de usuario que utiliza la aplicación. Si bien es cierto, un usuario podrá trabajar en el workspace que más le apetezca o incluso crear uno nuevo; pero siempre será interesante buscar el espacio de trabajo más recomendable según sea el uso que vayamos a dar de SmartSoft debido a los proyectos/ejemplos de gran utilidad que se ofrecen.

A continuación se presentan una serie de capturas de pantalla del entorno de desarrollo de SmartSoft, donde se describen las vistas más importantes así como la finalidad de cada una de ellas. Una vez seleccionado el workspace en el que vamos a trabajar (en este caso seleccionamos el *ws-examples*), arranca el toolchain. El siguiente paso es abrir un proyecto o crear uno nuevo. En este apartado, trabajaremos con los proyectos dados, dejando para el siguiente apartado el proceso de diseño desde cero.

En los proyectos-ejemplos proporcionados por el equipo de desarrollo de SmartSoft, podemos conocer la naturaleza del proyecto según sea su nombre:

- Proyectos con nombre ***"Comm*"***, se trata de un repositorio de objetos.
- Proyectos con nombre ***"Sample*"*** *o* ***"Smart*"***, se trata de un componente.
- Proyectos con nombre ***"Deploy*"***, se trata de un deployment.

Para la descripción del entorno de desarrollo, vamos a distinguir los proyectos en que se describen componentes, de los proyectos que describen deployments, ya que existen pequeñas diferentes entre uno y otro. Desde *Window/Show View* podemos acceder a las distintas vistas disponibles que posee el MDSD. La disposición de vistas utilizada que aparece en la figura 4.3 es la que se ha utilizado a lo largo de todo el proyecto. Se ha tomado como ejemplo el componente *SamplePushNewestClient*.

Lo primero que advertimos es que en el modelo, en la esquina superior derecha aparece una pieza de puzle. Esto es indicativo de que el proyecto es un componente. Veamos ahora la finalidad de cada una de las vistas:

- En la vista *Navigator* nos aparecen todos los proyectos que forman parte del workspace. Para cada uno de los proyectos podemos ver cuál es el conjunto completo de ficheros que lo forman. El modelo asociado al proyecto lo encontramos en */model/pim/'*_pim.di2'.*
- Desde la vista *Outline* podemos observar todos los elementos de los que se compone el modelo: puertos de comunicación del componente, las tareas o handlers que contiene, objetos de comunicación importados...

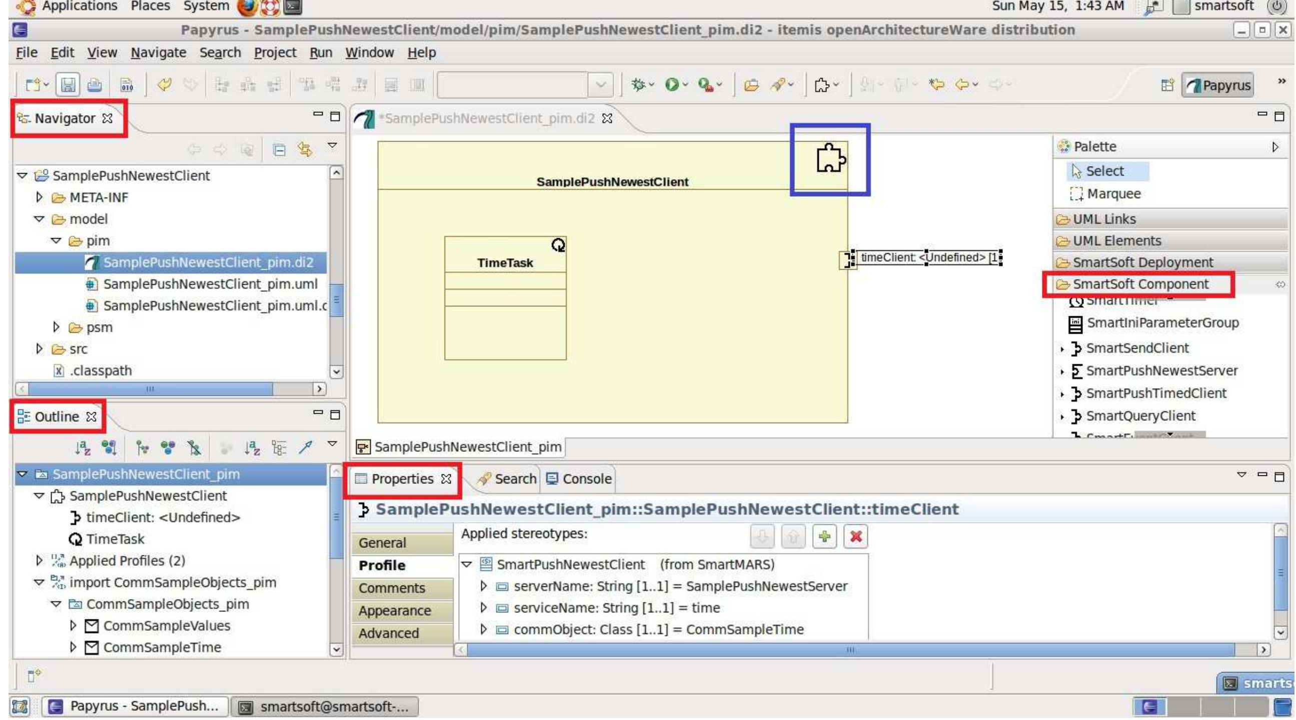


Figura 4.3: MDSD SmartSoft trabajando con un componente.

– En la vista *Properties*, como su propio nombre indica, encontramos las propiedades o configuración que posee el elemento seleccionado en ese momento.
– En la vista *Palette* tenemos todo lo necesario para la construcción de un diseño haciendo uso de SmartSoft. Dependiendo del tipo de proyecto, se hace uso de un apartado u otro.

Si por el contrario, el proyecto es un deployment:

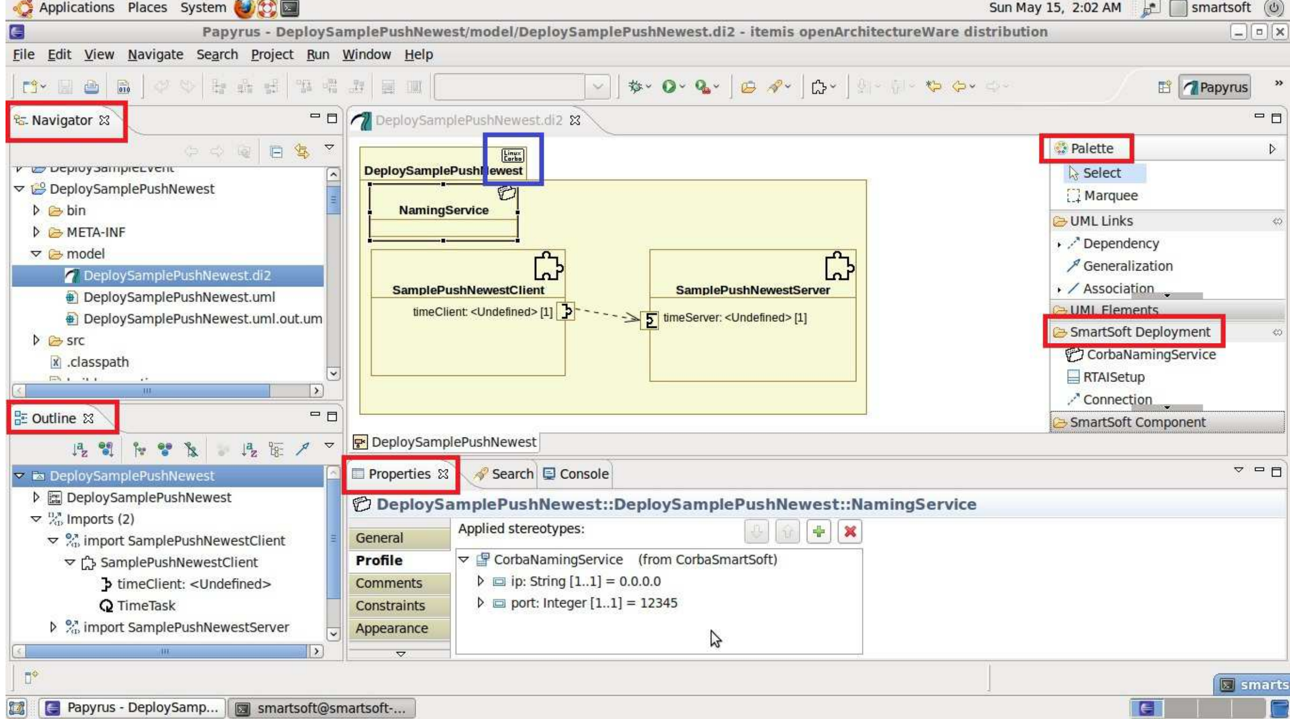


Figura 4.4: MDSD SmartSoft trabajando con un deployment

Vemos como si el proyecto es un deployment, ahora lo que aparece en la parte superior del modelo es un pequeño icono con "*Linux CORBA"* escrito en su interior. Las 4 vistas anteriormente descritas tienen la misma finalidad, pero orientadas a un uso con deployments. Dicho de otra manera, utilizando la misma vista, el uso que haremos de ella dependerá según sea el tipo de proyecto. Por ejemplo, mientras que en los componentes el apartado que utilizamos de la *Palette* es *SmartSoft Component*, en los deployments será *SmartSoft Deployment*. Todo esto se verá en detalle en el siguiente apartado.

4.3. Descripción del proceso de diseño utilizando el toolchain

Entendemos por tanto que la aplicación es el conjunto de estos tres elementos: el deployment, los componentes que forman parte de este, y los objetos de comunicación utilizados para la comunicación entre componentes. El proceso de diseño de una aplicación queda como la composición de las siguientes tres tareas:

- Construcción de los objetos de comunicación
- Construcción de los componentes
- Construcción de los deployments

En principio, cada una de estas tareas debe ser realizada por una persona distinta del equipo de desarrollo. El desarrollo basado por componentes distingue roles a lo largo del proceso de desarrollo y promueve su separación. En los siguientes apartados se detallan cada uno de estos puntos. Por último se describe el proceso con el que ejecutar la aplicación una vez construida.

4.3.1 Construcción de objetos de comunicación

Para la construir un objeto de comunicación, lo primero será disponer de un proyecto del tipo repositorio de objetos. En este tipo de proyecto solo podrán diseñarse objetos de comunicación, pudiendo contruir un número arbitrario de objetos de comunicación en él (esto no pasa con los componentes o deployments donde la relación es 1 proyecto por componente o deployment). A continuación se describe el proceso para la construcción de un objeto en un nuevo repositorio.

- *File / New / Project → Smart Robotic Project / SmartSoft Communication Object Repository* → Fijamos el nombre del repositorio.

De esta manera creamos un nuevo repositorio de objetos de comunicación. Para los diseños o proyectos realizados por los usuarios (dejando de lado los proyectos que proporciona el equipo de desarrollo de SmartSoft), el proyecto se ubica por defecto en la carpeta *"/home/<usuario>/ws-* "*, dependiendo así del workspace en el que el usuario haya iniciado el toolchain. Aún así, el constructor del objeto de comunicación es libre de seleccionar la ubicación del proyecto donde él desee. En apartado *"4.4 – Estructura de directorios de SmartSoft "* profundizaremos este aspecto de la ubicación de los distintos ficheros y proyectos según su tipo.

En la construcción de objetos de comunicación el apartado *SmartSoft CommObject* que aparece en la vista *Palette* es el que realmente interesa. Será aquí donde encontramos todo lo necesario para la construcción del objeto de comunicación. A su vez, este apartado está disponible debido al tipo de proyecto que hemos especificado (*SmartSoft Communication Object Repository),* apareciendo este apartado en la *Palette* y no otros, como veremos en los procesos de diseño de componentes o deployments.

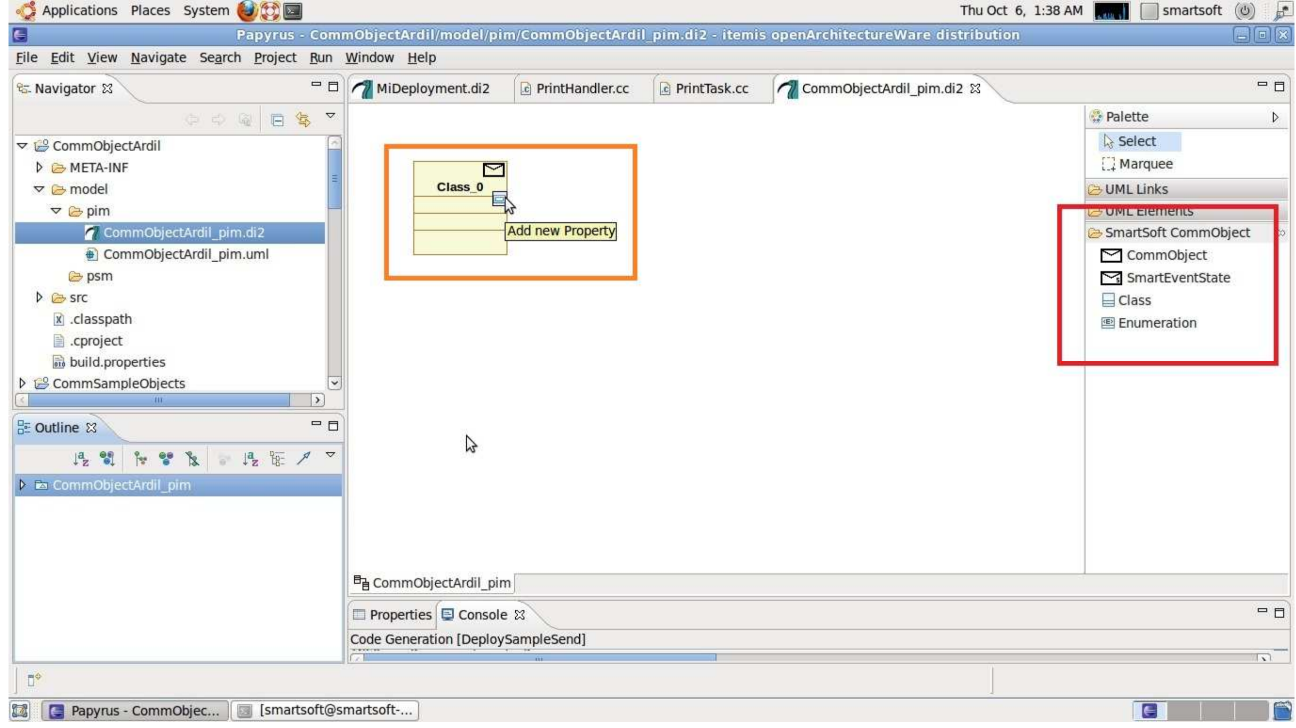


Figura 4.5: Aspecto de un proyecto del tipo repositorio de objetos de comunicación y la vista Palette.

Modelado del Objeto de Comunicación

Pasamos ahora a modelar el objeto propiamente dicho. Para ello, hacemos clic con el botón izquierdo del ratón sobre *CommObject* en la *Palette*, y clic de nuevo sobre la vista general del toolchain. De esta manera tendremos algo como la figura 4.6. A continuación pasamos a cambiarle el nombre al objeto (no confundir con el nombre que se ha especificado anteriormente que era del repositorio de objetos):

- *Seleccionado el objeto / Vista Properties / General / Name.*

El siguiente paso es establecer los tipos de datos de los que se compone el objeto, que pueden ser desde un entero de 8 bits, hasta un registro de otros objetos de comunicación (ver apartado 3.2.1). Para ello hacemos lo siguiente:

- En el propio modelo, dejamos el ratón sobre la sección que queda justo debajo del nombre del objeto de comunicación, hasta que aparezca la opción *Add new Property.* Hacemos clic con el botón izquierdo del ratón.

A continuación tenemos que especificar el tipo de propiedad del que se trata:

- *Seleccionar objeto de comunicación / vista Properties / General / Value Definition / Type*.

Pulsando el símbolo con el signo *"+"*, nos aparecerán los posibles valores que podemos asignarle a la propiedad. En *SMARTMARS_PROFILE* aparecerán los tipos de datos primitivos (entero con signo y sin signo de diferentes tamaño, float y double, boolean, strings…). En la figura 4.6, se muestran las distintas ventanas que utilizamos para llevar a cabo esta tarea.

Repitiendo todo este proceso podemos añadir tantos objetos de comunicación como queramos a nuestro repositorio de objetos.

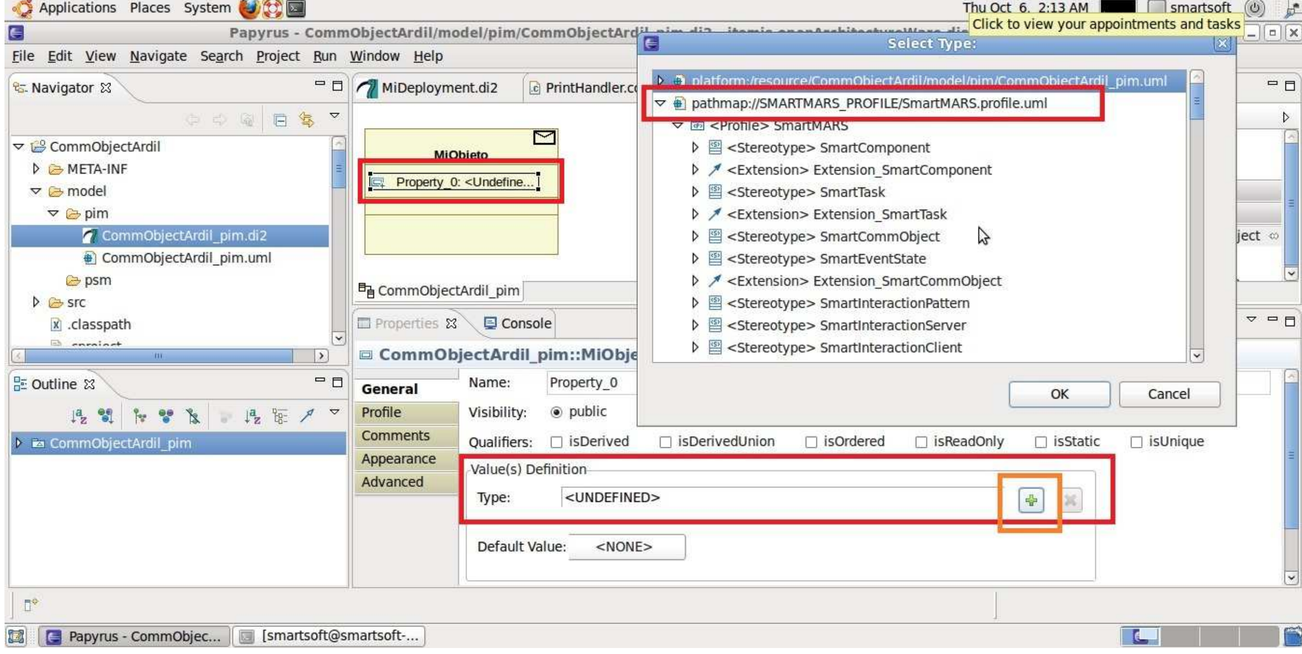


Figura 4.6: Construcción de un objeto de comunicación mediante el toolchain

Generación del código asociado

Una vez definida la estructura de datos del objeto, generamos el código asociado al modelo. Para ello, desde la vista *Navigator* hacemos lo siguiente:

- *Botón derecho sobre el nombre del proyecto / Smart Robotics / Run Code Generator*

Una vez finalizado este proceso, en la carpeta *$SMART_ROOT/src/nameProject/gen* se crean los ficheros *".hh"* y *".cc"* asociados al objeto de comunicación (de igual nombre que el objeto). Además en esta carpeta se genera también el *".idl"*.
Si accedemos a ellos, como ya se comentaba en el apartado 3.2.1, en estos ficheros aparece la interfaz del framework que ha sido generada automáticamente por el toolchain (métodos *identifier(...)* , *get(...),* y *set(...)).* De la misma manera, en estos ficheros el constructor del objeto implementará las funciones miembro que considere oportunas y que definen la interfaz de usuario del objeto de comunicación.

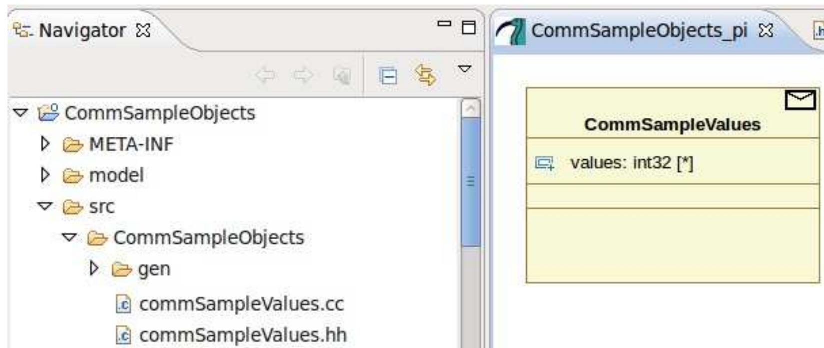


Figura 4.7: Ficheros generados con el proceso Run Code Generator en un objeto de comunicación.

Build Project

Por último solo queda realizar el *build project*, con lo que el objeto de comunicación construido estará listo para ser importado y posteriormente utilizado por los componentes que así lo requieran, dando por finalizado el primer paso en la construcción de una aplicación.

- *Botón derecho sobre el nombre del proyecto / Smart Robotics / Build Project*

4.3.2. Construcción de componentes

El siguiente paso en el proceso de diseño de una aplicación es la construcción de los componentes. Los componentes hacen uso de los objetos de comunicación para intercambiar información con otros componentes a través de los patrones de comunicación establecidos. Son cada una de las piezas del puzle que se compondrá en el posterior deployment.

Para crear componentes con el toolchain se darán los siguientes pasos:

- ***Crear proyecto del tipo componente***
- ***Importar objetos de comunicación***
- ***Realización del modelo (incorporar puertos, tareas, handlers…)***
- ***Generar código asociado al modelo***
- ***Completar Componentes***
- ***Build Project***

Crear proyecto del tipo SmartSoft Component

El primer paso será crear un proyecto de tipo componente. En este tipo de proyecto solo podrán diseñarse componentes (un componente por proyecto), y se denomina: *Smart Robotic Project / SmartSoft Component.* Debido a que en este tipo de proyectos solo es possible diseñar un componente, el componente toma directamente por defecto el nombre del proyecto. Los proyectos realizados por usuarios se ubican por defecto en la carpeta *"/home/<usuario>/ws-* "*, dependiendo así del workspace en el que el usuario haya iniciado el toolchain. Aún así, el constructor del componente es libre de seleccionar la ubicación del proyecto donde él desee.

- *File / New / Project → Smart Robotic Project / SmartSoft Component* → Fijamos el nombre del proyecto/componente

Importar objetos de comunicación

Los objetos de comunicación son los datos con los que se realiza la comunicación entre componentes. Representan el contenido transmitido a través de los puertos de comunicación. Aquellos objetos que utiliza nuestro componente, deben ser importados al proyecto actual. Más concretamente importamos el repositorio de objetos donde está contenido.

- *Botón derecho sobre la vista Outline / Smart Robotics / Import CommObject Repository.*.Seleccionamos entonces el repositorio

En ocasiones este proceso no se realiza correctamente, dando lugar a situaciones en la que parece imposible importar los componentes aun habiendo realizado el proceso descrito. Los motivos parecen ser un problema de Papyrus, y la solución pasa por cerrar el modelo del proyecto y volver abrirlo. De esta manera, si efectivamente hemos importado bien los objetos, aparecerán con seguridad.

Realización del modelo

Para llevar a cabo las tareas de modelado como ya ocurría anteriormente, es de especial interés la vista *Palette,* y en este caso concreto el apartado *SmartSoft Component.* Será aquí donde encontramos tod999o lo necesario para la construcción del componente. Este apartado está ahora disponible debido al tipo de proyecto con el que estamos trabajando. Existe por tanto una correspondencia entre los apartados de la *Palette* y el tipo de proyecto (no es posible ahora realizar un objeto de comunicación como hacíamos en el apartado anterior).

Aquí encontramos todos los patrones de comunicación que ofrece SmartSoft (clientes y servidores), así como los handlers asociados a aquellos patrones que lo poseen (en el caso del patrón *Send* el servidor tiene un handler; sin embargo en *PushNewest* no existe handler alguno). También aparecen tareas *SmartTask,* semáforos *SmartMutex*, temporizadores *SmartTimer,* o estructuras de datos del tipo *SmartIniParameterGroup* donde es posible definir variables o parámetros globales desde el punto de vista de las tareas y handlers del componente, todo ello desde el modelo.

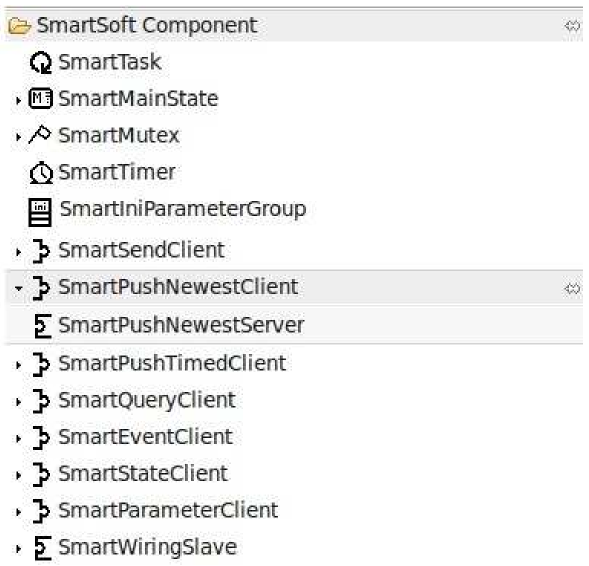


Figura 4.8: Apartado SmartSoft Component de la vista Palette

En la construcción de un componente, un paso clave es la correcta configuración de los puertos de comunicación. Cada uno de los puertos que posee SmartSoft, tiene unos campos específicos en el apartado *Profile*. Aquí se determinará el handler asociado a un puerto, el nombre del servicio que el cliente ofrece o requiere, el objeto de comunicación utilizado, etc.

Para un mismo patrón de comunicación también existe distinción entre el cliente y el servidor. Volviendo al ejemplo del patrón *Send,* el puerto de comunicación del servidor tendrá en el *Profile* un campo en que especificar cuál es el handler asociado a dicho puerto; campo que no tendrá el servidor.
Otro ejemplo es el campo *Cycle* que posee únicamente el puerto del servidor del patrón *PushTimed*, donde se especifica el tiempo entre envíos.

Como se ha comentado, en el proceso de configuración de los puertos es necesario especificar los objetos de comunicación que van a utilizarse para la comunicación con otros componentes (de ahí a que sea necesario importar el repositorio de objetos anteriormente). En la figura 4.9 se destacan las partes más importantes en el proceso de modelado.

Vemos como a la derecha de la figura aparece la *Palette* con el conjunto de patrones de comunicación clasificados por tipos. En la pestaña *Outline* aparecen todos los elementos que forman parte del proyecto, tanto elementos añadidos desde la *Palette*, como elementos importados (en caso de los componentes los elementos importados serán objetos de comunicación). En la parte inferior aparece el *Profile* del elemento seleccionado en ese momento, que para el caso concreto de la figura es el puerto *SmartSendServer.*

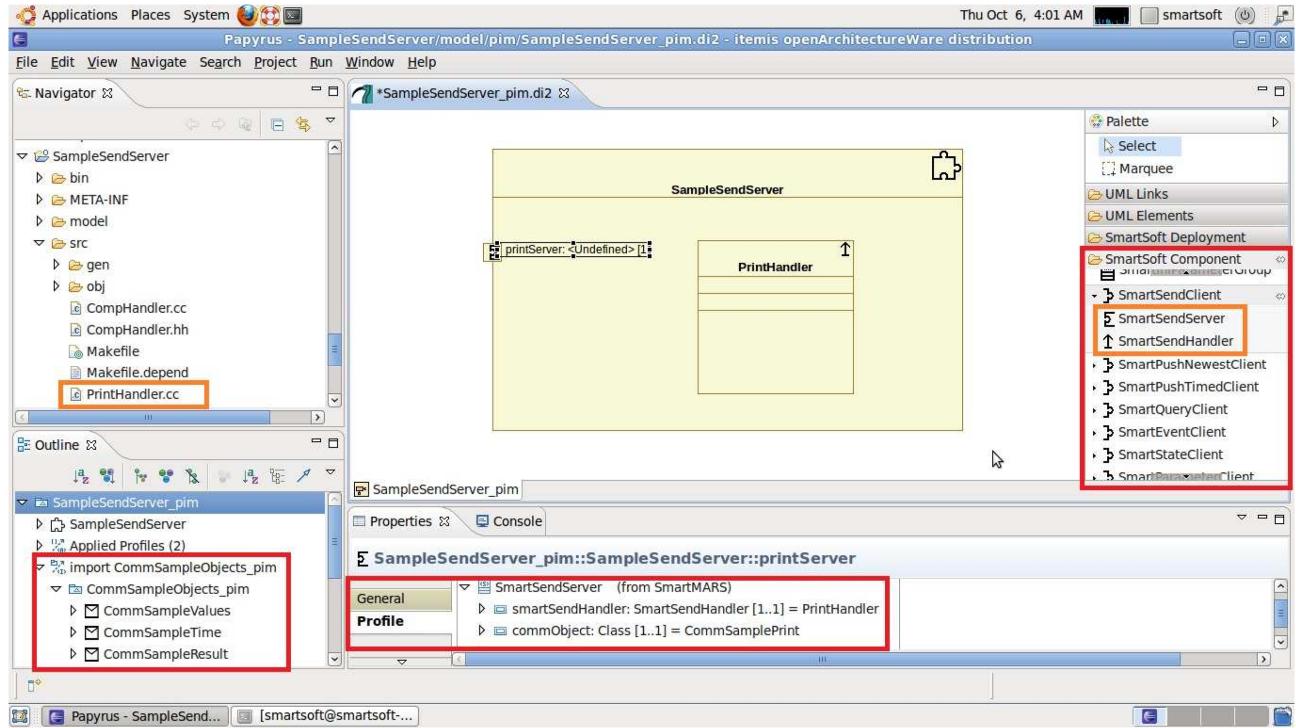


Figura 4.9: Construcción de un componente mediante el toolchain

Algunas de las propiedades que poseen los puertos de comunicación en su perfil podrían resultar contradictorias con el enfoque basado en componentes que dice tener SmartSoft. Un ejemplo de esto es la propiedad ServerName que tiene un cliente del tipo Send, o el nombre del servicio al que se suscribirá con dicho puerto.

"¿Por qué debe conocer el constructor del componente cliente ese tipo de información?" Como era de esperar, no tiene por qué hacerlo. "¿Para qué esas propiedades entonces?" La razón de la existencia de estos campos es debido a que es posible ejecutar un componente sin la obligación de un deployment superior. La configuración de los puertos se toma de estas propiedades en lugar de hacerlo del deployment. En situaciones de t*esting* de un componente, podría resultar farragoso la construcción de un deployment si por ejemplo en la aplicación solo van a intervenir 2 componentes queriendo así ahorrar recursos. Aún así una vez testeado, el deployment acabará realizándose y no tendrá en cuenta estas propiedades.

Generar código asociado al modelo

Finalizado el proceso de modelado del componente, generamos el código fuente asociado al modelo. Se realiza de igual manera a como hacíamos en el diseño de los objetos de comunicación:

- *Botón derecho sobre el nombre del proyecto / Smart Robotics / Run Code Generator*

Completar Componentes

De todos los ficheros generados, el constructor del componente únicamente tratará los ficheros "*.cc*" que así lo requieran, y completar los denominados "*hotspots"*. Estos ficheros de código en los que el constructor de componentes tendrá que intervenir son aquellos que están vinculados a tareas *SmartTask* o a *handlers* de puertos de comunicación. Aquí se define el comportamiento propio del componente, así como el tratamiento de los objetos de comunicación y la gestión de los servicios que realiza el componente tanto como proveedor como suscriptor.

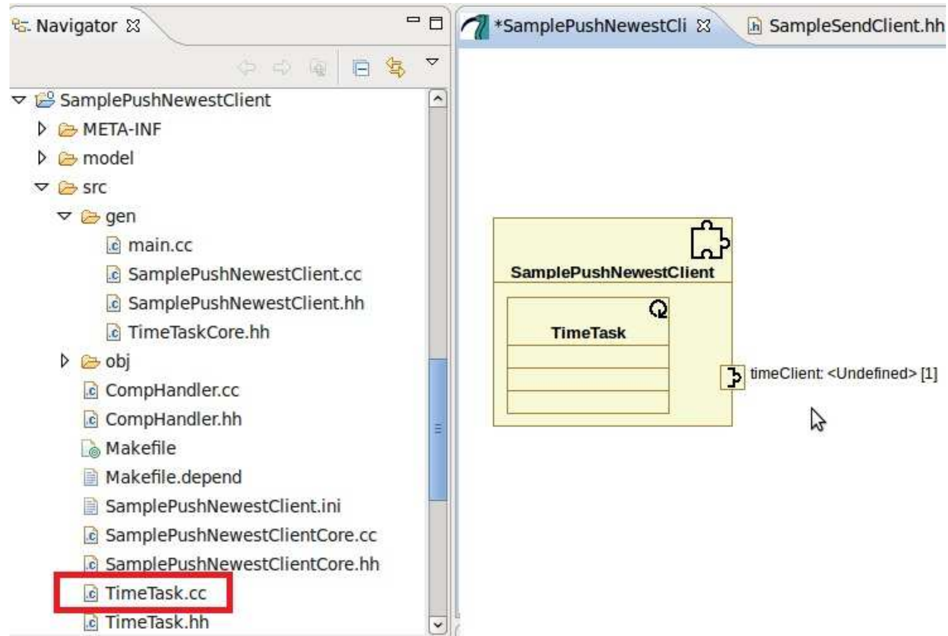


Figura 4.10: Ficheros generados con el proceso Run Code Generator en un componente.

Para este ejemplo en concreto done el componente posee una tarea denominada *TimeTask*, el constructor del componente tendrá que implementar en el fichero *TimeTask.cc* el comportamiento que tendrá el componente.

El acceso a los elementos definidos en el modelo del componente (ya sean puertos de comunicación, mutex o IniParameterGroup) se puede realizar de las siguiente maneras:

```
COMP->nombreElemento;
NombreComponente::instance()->nombreElemento;
```

Para el caso concreto de la figura 4.10, en el fichero asociado a la tarea TimeTask para acceder al puerto timeClient que es del tipo SmartPushNewestClient, se haría de la siguiente forma:

```
COMP->timeClient->getUpdateWait(a);
```

, siendo el método *getUpdateWait* es una función miembro del cliente del patrón *PushTimed* (ver apartado 3.3.4 – Patrón PushTimed). Cabe destacar que en los ficheros *main.cc* y *SamplePushNewestClient.cc*, se encuentra el código encargado de crear u lanzar el componente (ver apartado 3.3.2).

Build Project

Por último se realiza el *build project.* De esta manera el componente construido queda terminado, genera en la carpeta *"src/obj"* los ficheros *".o"* necesarios, y está listo para que en un futuro deployment se haga uso de él.

- *Botón derecho sobre el nombre del proyecto / Smart Robotics / Build Project*

El ejecutable del componente que resulta al realizar este proceso de build, se encuentra en un directorio diferente al del proyecto. A este directorio aparte, van a parar todos los ejecutables de los componentes (ya sean componentes proporcionados por el equipo de desarrollo de SmartSoft, o componentes construidos por usuario). En el apartado 4.4 se detalla la estructura de ficheros que sigue SmartSoft y como se ubican todos los elementos.

Con la construcción del componente se da por finalizado el segundo paso en la construcción de una aplicación.

4.3.3. Construcción de deployments

El último paso en el proceso de diseño de una aplicación es la construcción del deployment. Los componentes antes generados y que van a formar parte de la aplicación, se importarán al proyecto asociado al deployment. Podemos hacer el símil con un puzle: cogiendo las piezas del puzle (serían los componentes), este sería el tablero donde colocarlas y distribuirlas de forma que el puzle 'cuadre' y se alcance la solución buscada.

Crear un proyecto de tipo Deployment

Lo primero será crear un proyecto del tipo deployment. Como ya pasaba en la construcción de los componentes, este tipo de proyectos solo puede contener un único deployment, que por defecto toma el nombre del proyecto en el que está contenido.

- *File / New / Project* → *Smart Robotic Project / SmartSoft Deployment* → Fijamos el nombre del proyecto/deployment

Como comentábamos en apartados anteriores, los proyectos realizados por usuarios (dejando de lado los proyectos que proporciona el equipo de desarrollo de SmartSoft), se ubican por defecto en la carpeta *“/home/<usuario>/ws-*”*, dependiendo así del workspace en el que el usuario haya iniciado el toolchain. Aún así, el constructor del deployment es libre de seleccionar la ubicación del proyecto donde él desee.

Importar componentes

El siguiente paso será importar al proyecto los componentes que van a formar parte del deployment: Lo realizamos de forma muy parecida a cuando importábamos los objetos a los componentes:

- Desde la vista Outline *Botón derecho sobre 'DeploySampleSend ' / Smart Robotics / Import Component.*

Al igual que ocurría cuando importábamos los objetos de comunicación, para asegurarnos de que todo el proceso se ha realizado correctamente, debemos cerrar y volver a abrir el proyecto.

Composición del modelo

Una vez importado el componente, desde la pestaña *Outline*, tendremos que pinchar con el botón izquierdo del ratón sobre él, y arrastrarlo hasta el deployment. De esta manera se incluye el componente al deployment.

En el modelo del componente que aparece en el deployment, no aparecerán por defecto los elementos que forman de él (puertos, tareas, etc). Esto radica de un problema de Papyrus. El constructor del deployment tendrá que arrastrar aquellos puertos de comunicación del componente que van a comunicarse con otros (no es necesario incorporar al modelo las tareas, handlers, puertos no utilizados...), y que más tarde serán vinculados.

Además, al crear un proyecto del tipo *SmartSoft Deployment*, en la vista *Palette* nos aparece un apartado específico para el modelado de deployments: *SmartSoft Deployment.* Es de especial interés el *NamingService,* elemento que todos los deployment tienen que tener y gracias al cual los componentes pueden interactuar entre sí.

En *Properties / Profile* del *NamingService,* tendremos que especificar la dirección IP y puerto donde se encuentra dicho servicio. Para la realización de este PFC siempre se ha utilizado la dirección localhost (0.0.0.0), y un puerto cualquiera como puede ser el 12345.

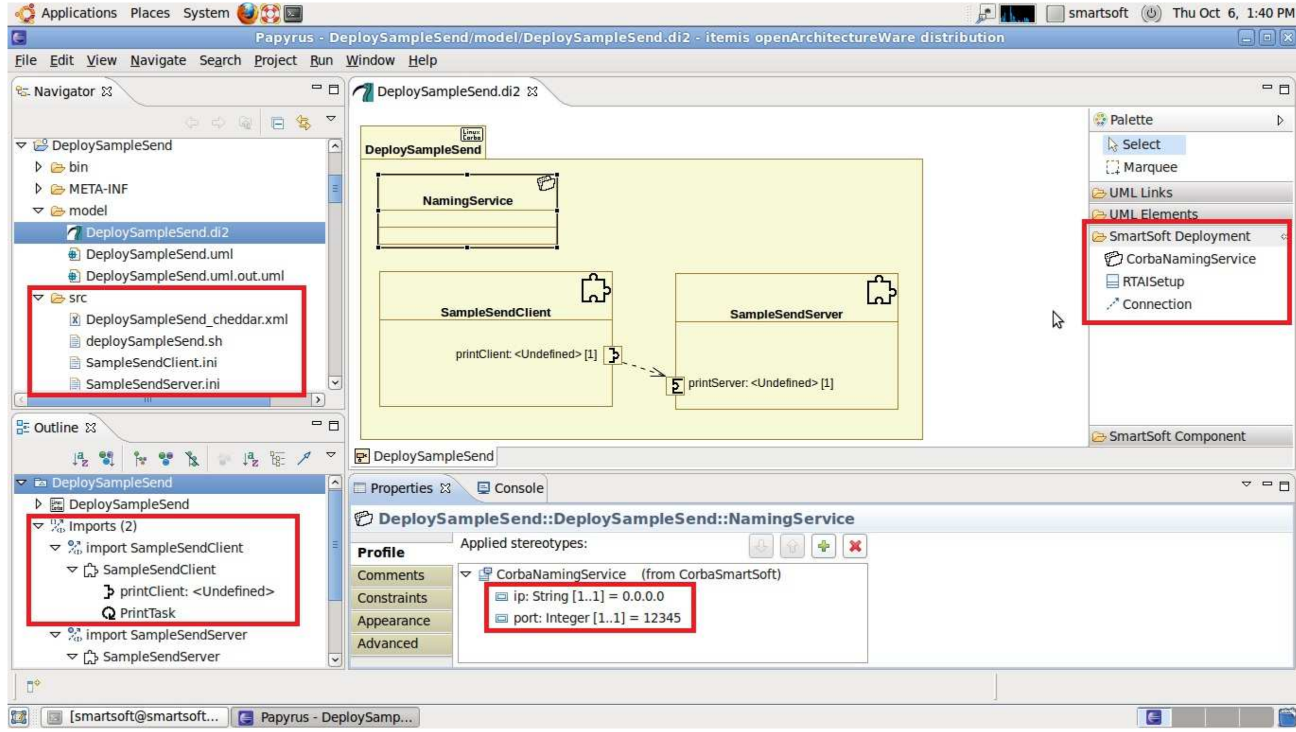


Figura 4.11: Construcción de un deployment mediante el toolchain

Conexión de componentes

El siguiente paso es relacionar los componentes entre sí. Para ello hacemos uso de *'Connection'*, que se encuentra también en el apartado *SmartSoft Deployment* dentro de la vista *'Palette'*. De esta manera ligamos unos componentes a otros a través de los puertos de comunicación.

El constructor de un componente no conoce el uso que hará de éste el constructor del deployment. Esto produce que algunas de las propiedades de los puertos de comunicación como puede ser el nombre del servidor para un puerto cliente, no sean correctas y deban ser modificados. Siempre prevalecerá aquello que se haya especificado en el modelo del deployment.

Proceso de Deploy

Completado todo el proceso de modelado, solo queda realizar el proceso de *Deploy* del proyecto:

- *Botón derecho sobre el proyecto desde la vista Navigator / Smart Robotics / Deploy.*

Tras realizar esta operación de *Deploy* sobre el deployment, se crea dentro de la carpeta *src* correspondiente al proyecto, un script (fichero *".sh")* denominado de igual manera que el deployment. En el siguiente apartado se especificará el contenido de este fichero.

De esta manera damos por finalizado todo el proceso de diseño de una aplicación, partiendo de la construcción de los objetos de comunicación, construyendo los componentes según las necesidades del usuario y que hacen uso de los objetos de comunicación; y el deployment final donde especificar las relaciones que existen entre los componentes.

4.3.4. Ejecución de la aplicación

Para la ejecución de una aplicación tenernos 2 posibilidades:

- Ejecutar la aplicación por medio de un deployment

  Esta será la forma natural de hacerlo. Para ello haremos uso del script que se genera en el proceso de *Deploy* del deployment. Este script se encarga de poner en marcha el *NamingService* de CORBA, así como de lanzar cada uno de los componentes que forman parte del proyecto. La manera en la que los componentes se comunicarán es la que se haya fijado en el deployment. Los ficheros "*.ini*" de los componentes son generados también en esta misma carpeta y necesarios para la ejecución.

  Para ejecutar la aplicación escribimos el siguiente comando:

  ```
  >> deployment.sh start
  ```

  Para detener la ejecución:

  ```
  >> deployment.sh stop
  ```

- Ejecutar los componentes por separado sin la implicación de un deployment.

  También es posble ejecutar los componentes "a mano" uno a uno. Esto puede ser de utilidad en procesos de testing donde el número de componentes con el que trabajamos es reducido, y la construcción de un deployment para realizar esa prueba se hace algo farragoso.

  La comunicación entre los componentes viene fijada por la configuración de los puertos, al contrario de cómo ocurre en el caso anterior (algo lógico ya que ahora no hay deployment).

  Si queremos que los componentes ejecutados se comuniquen entre sí, ahora tenemos que asegurarnos de que el NamingService de CORBA se está ejecutando correctamente. Para ello hacemos uso de los ejecutables *ShowCORBA, StartCORBA* y *StopCORBA*. Este proceso en caso de utilizar un deployment, lo realiza automáticamente el script *".sh"*.

  Para lanzar un componente debemos ir a la carpeta donde se encuentra el ejecutable (en el siguiente apartado se describe como estructura los ficheros SmartSoft), y lanzar aquel que tiene el mismo nombre del componente.
  Parar ejecutar el componente en una nueva terminal:

  ```
  >> # xterm -e exampleComponent &
  ```

4.4. Estructura de directorios de SmartSoft

En este apartado se muestra como se estructuran los distintos elementos de SmartSoft en el sistema de ficheros del sistema. Lo haremos utilizando el comando de linux *tree*, que lista los ficheros de un directorio. Se indican aquellos que son de especial interés.

Empezamos por el directorio home del usuario:

```
smartsoft@smartsoft-vm:~$ pwd
/home/smartsoft
smartsoft@smartsoft-vm:~$
smartsoft@smartsoft-vm:~$ ls
Desktop  Downloads  SOFTWARE  src  ws-examples  ws-mdsd  ws-toolchain  zafh.jpg
smartsoft@smartsoft-vm:~$
smartsoft@smartsoft-vm:~$ tree -d -L 4 ws-examples/
ws-examples/
|-- MiComponente
|   |-- META-INF
|   |-- model
|   |   |-- pim
|   |   `-- psm
|   `-- src
|       |-- gen
|       `-- obj
`-- MiObjeto
    |-- META-INF
    |-- model
    |   |-- pim
    |   `-- psm
    `-- src
        |-- MiObjeto1
        `-- MiObjeto2
```

Vemos como aparecen los directorios de los distintos workspaces: **ws-examples, ws-mdsd** y **ws-toolchain.** En estas carpetas encontramos para cada workspace, las implementaciones de los diseños que el usuario ha construido, ya sean objetos de comunicación, componentes o deployments. Aparece el diseño de un componente y de un repositorio de objetos. Vamos a listar todos los ficheros del componente "MiComponente":

```
smartsoft@smartsoft-vm:~$ tree -C -L 3 ws-examples/MiComponente/
ws-examples/MiComponente/
|-- build.properties
|-- META-INF
|-- MiComponente.recipes
|-- model
|   |-- pim
|   `-- psm
`-- src
    |-- CompHandler.cc
    |-- CompHandler.hh
    |-- gen
    |   |-- main.cc
    |   |-- MiComponente.cc
    |   |-- MiComponente.hh
    |   `-- MitareaCore.hh
    |-- Makefile
    |-- MiComponenteCore.cc
    |-- MiComponenteCore.hh
    |-- MiComponente.ini
    |-- Mitarea.cc
    |-- Mitarea.hh
    `-- obj
        |-- CompHandler.o
        |-- MiComponenteCore.o
        `-- Mitarea.o
```

Observamos como dentro de la carpeta *src* del componente, tenemos el fichero asociado a la tarea *"MiTarea.cc*", donde el constructor del componente ha definido el comportamiento del mismo.

A continuación vemos como se estructuran los proyectos que son proporcionados por el equipo de desarrollo de SmartSoft. Como aparece en el apartado que detalla el proceso de instalación (ver apartado -), SmartSoft realiza la instalación a partir del directorio *$SMART_ROOT/src*. Listando únicamente los directorios obtenemos lo siguiente:

```
smartsoft@smartsoft-vm:~$
smartsoft@smartsoft-vm:~$ echo $SMART_ROOT
/home/smartsoft/SOFTWARE/smartsoft
smartsoft@smartsoft-vm:~$
smartsoft@smartsoft-vm:~$ tree -d -C -L 3 $SMART_ROOT
/home/smartsoft/SOFTWARE/smartsoft
|-- bin
|-- data
|-- doc
|-- etc
|-- include
|   |-- CommBasicObjects
|   `-- ...
|-- lib
`-- src
    |-- components
    |   |-- SmartLaserObstacleAvoid
    |   |-- SmartPioneerBaseServer
    |   |-- SmartPlayerStageSimulator
    |   `-- ...
    |-- deployments
    |   |-- DeployLaserObstacleAvoid
    |   `-- ...
    |-- example-components
    |   |-- SamplePushTimedClient
    |   `-- ...
    |-- example-deployments
    |   |-- DeploySamplePushNewest
    |   `-- ...
    |-- interfaceClasses
    |   |-- CommBasicObjects
    |   `-- ...
    |-- smartSoftKernel
    |   |-- smartExampleComponent1
    |   `-- ...
    `-- utility
        |-- armadillo
        `-- ...
```

En la carpeta *src/example-components* se encuentran las implementaciones de los componentes que son proporcionados por el equipo de SmartSoft y que aparecen al iniciar el workspace *ws-examples* (ver apartado 4.2). La carpeta *src/example-deployments* contiene las implementaciones de los deployments de estos componentes y que también aparecen al iniciar el MDSD Toolchain en este workspace.

Por otro lado, en la carpeta *src/components* se encuentran las implementaciones de los componentes del workspace *ws-mdsd* (ver apartado 4.2). La carpeta *src/deployments* contiene las implementaciones de los deployments de estos componentes,que aparecen al iniciar el workspace *ws-mdsd.*

En el directorio */src/interfaceClasses* se encuentran las implementaciones de los repositorios de los objetos de comunicación que facilita el equipo de SmartSoft. Los repositorios que encontramos en este directorio, se encuentran repartidos entre el workspace *ws-examples* y el *ws-mdsd*.

En la carpeta */bin* se almacenan los ejecutables de todos los componentes, ya hayan sido diseñados por los usuarios y se encuentren en *"/ws-*",* sean componentes de la carpeta */src/components* o */src/example-components.* Los script resultantes de los procesos de Deploy, siempre buscarán en este directorio los ejecutables de los componentes.

```
smartsoft@smartsoft-vm:~/SOFTWARE/smartsoft$ ls
bin  data  doc  etc  include  lib  LICENSE  README  showCORBA  src  startCORBA  stopCORBA
```

En la carpeta *$SMART_ROOT,* también encontramos los ejecutables *showCORBA, statCORBA, y stopCORBA;* necesarios para el correcto funcionamiento de *NamingService de CORBA.*

5. EJEMPLOS BÁSICOS

5.1. Ejemplo de utilización patrón “Send”

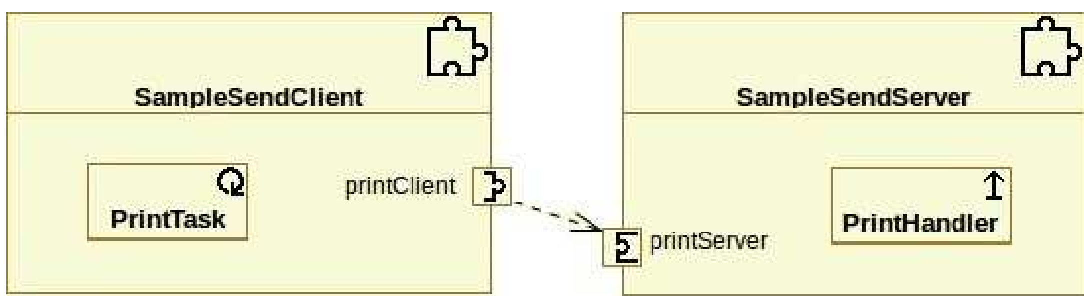


Figura 5.1: Diagrama de funcionamiento ejemplo Send

En este apartado se describe el proceso de construcción de una aplicación sencilla, en la que mostrar el funcionamiento del patrón de comunicación “Send”. Se construyen dos componentes: un cliente y un servidor. El cliente prepara y envía repetidamente una cadena de texto al servidor. Éste por su parte se limita a recibirlas e imprimirlas sin más. El objeto de comunicación utilizado es importado del proyecto *CommSampleObject.*

El componente servidor se denomina *SampleSendServer.* Este componente tiene un puerto del tipo *SmartSendServer* a través del cual recibe el objeto de comunicación con el que trabaja. Además posee un handler del tipo *SmartSendHandler* donde realizar el tratamiento del objeto de comunicación *CommSamplePrint* limitándose su tratamiento a una impresión por pantalla. Para ello utiliza la función *print* propia del objeto de comunicación.

Por otra parte, tenemos el componente *SampleSendClient* con un puerto del tipo *SmartSendClient* y una tarea *SmartTask*. La tarea se encarga por un lado de preparar el mensaje (*CommSamplePrint*), mediante la función “*set”* de estos, para más tarde haciendo uso de la función “*send”* del patrón de comunicación, enviar el objeto al servidor.

Construidos los dos componentes, se realiza un deployment denominado *DeploySampleSend* en el que relacionar ambos componentes. Dividimos por tanto el diseño en tres secciones:

- Construcción del componente *SampleSendServer.*
- Construcción del componente *SampleSendClient.*
- Construcción y ejecución del deployment *DeploySampleSend.*

Construcción del componente servidor “SampleSendServer”

Creamos un nuevo proyecto de tipo Componente:

- *File / New / Project* → *Smart Robotic Project / SmartSoft Component* → Fijamos el nombre del componente: '*SampleSendServer*'

Abrimos el modelo. Para ello, buscamos en la pestaña *Navigator* el proyecto recién creado:

- *SampleSendServer / model / pim / SampleSendServer_pim.di2*

Este componente está formado por el puerto de comunicación del servidor *SmartSendserver* y un handler asociado del tipo *SmartSendHandler.* Estos dos elementos los encontramos en el apartado *SmartSoft Component* dentro de la vista *Palette*.

Para hacerlos formar parte de nuestro diseño, para cada uno de ellos debemos hacer clic con el botón izquierdo de nuestro ratón en el elemento de la paleta, y a continuación clic de nuevo ahora sobre el modelo. Una vez dichos elementos forman parte del modelo, es posible recolocarlos para una mejor visualización a gusto del usuario manteniendo pulsado el botón izquierdo.

Una vez añadidos estos elementos vamos a modificar sus nombres. Esto lo podemos hacer en las *Properties / General* de cada elemento. Los renombramos como: *printServer* y *PrintHandler* respectivamente (se recomienda que los puertos tengan la primera letra en minúscula y de esta manera evitaremos un "Warning" en el proceso de generación de código).

A continuación vamos a importar al espacio de trabajo el proyecto *CommSampleObject*. En este repositorio se incluyen objetos de comunicación básicos como es el *CommSamplePrint*.

Es posible importar únicamente el objeto que vayamos a utilizar en nuestro diseño en lugar de hacerlo con el proyecto o repositorio completo. Sin embargo, debido a que han surgido algunos errores cuando se han importado objetos sueltos, y a la comodidad que supone importar el proyecto completo, decidimos realizarlo de este modo.

Lo hacemos de la siguiente manera:

- En la pestaña Outline: *Botón derecho sobre 'SampleSendServer' / Smart Robotics / Import CommObject Repository.* A continuación buscamos el *'CommSampleObject'*.

Para comprobar que el proceso se ha realizado correctamente, debido a un problema de SmartSoft, será necesario cerrar el proyecto y volver abrirlo.

El siguiente paso será proceder a configurar el puerto. Para ello accedemos al perfil del mismo: *Properties / Profile.* Fijamos los siguientes valores:

– *smartSendHandler: smartSendHandler [1..1] = PrintHandler*
– *commObject : Class [1..1] = CommSamplePrint*
– *serviceName : String [1..1] = print*

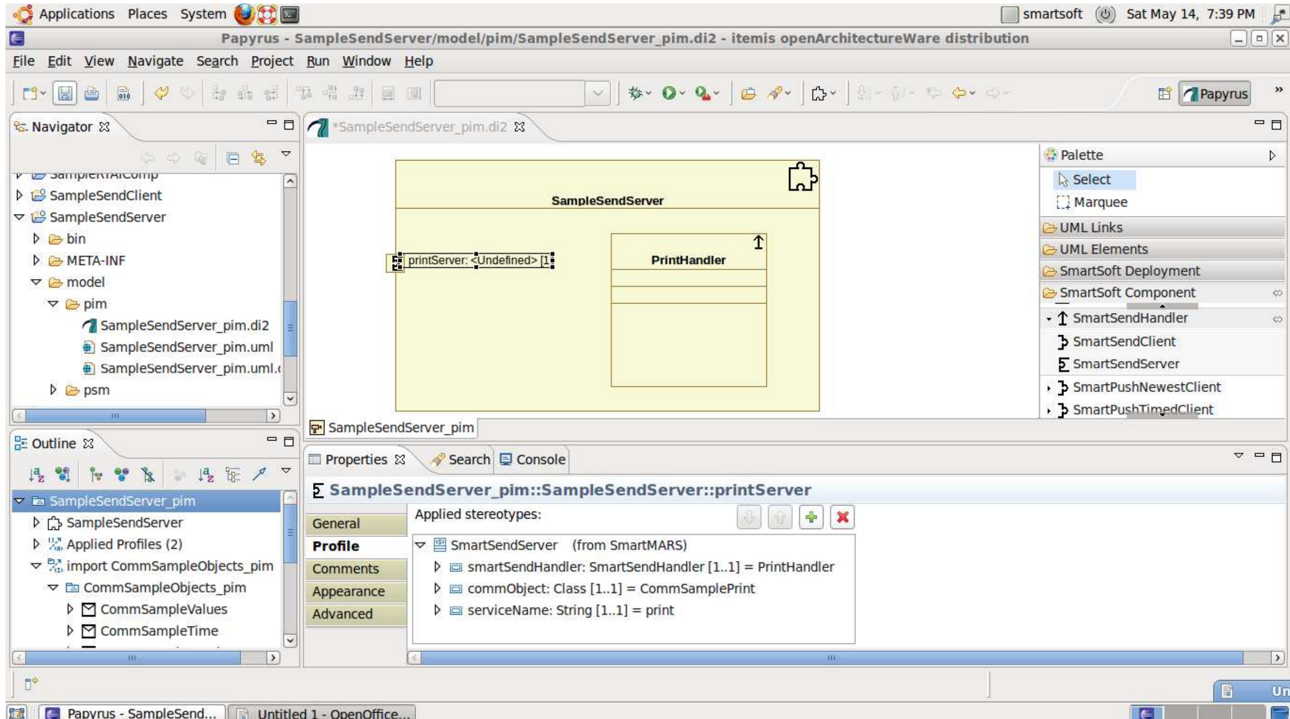


Figura 5.2: Modelado del componente SampleSendServer

Vemos de esta manera, como se vincula el puerto de comunicación con el handler; así como la manera en la que se define el tipo de objeto de comunicación que utiliza el mismo. Por último establecemos el identificador que tendrá el servicio establecido entre los componentes.

El último paso que realizamos con el modelo es generar el código asociado al diseño. Para ello, hacemos lo siguiente:

- *Botón derecho sobre el proyecto en la vista Navigator / Smart Robotics / Run Code Generator*

Tras haber generado el código asociado, en la carpeta *src* de nuestro proyecto aparecen distintos ficheros. Nosotros estamos interesados en el *PrintHandler.cc*. En este fichero se establece el comportamiento del handler asociado al puerto *SampleSendServer.*

El código que establecemos es el siguiente:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
void PrintHandler::handleSend(const CommSampleObjects::CommSamplePrint &r) throw(){
    std::cout << "Lo que he recibido es: ";
    r.print();
}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Por tanto en la ejecución de dicho componente se imprimirá la cadena: *“Lo que he recibido es:”,* seguido del objeto de comunicación (otro string) recibido por el puerto de comunicación que es enviado desde el cliente.

De esta manera hemos finalizado todo el proceso de diseño del componente *SampleSendServer*. Ya solo nos queda realizar el 'Build' sobre el proyecto:

- *Botón derecho sobre el proyecto desde la vista Navigator / Smart Robotics / Build Project*

Construcción del componente cliente “SampleSendClient”

Creamos un proyecto de tipo componente:

- *File / New / Project → Smart Robotic Project / SmartSoft Component →* Fijamos el nombre del componente. ('*SampleSendClient*')

Abrimos el modelo:

- *Desde la vista Navigator / SampleSendClient / model / pim / SampleSendClient_pim.di2*

Este componente está compuesto por un puerto *SmartSendClient* y una tarea *SmartTask.* Ambos elementos los encontramos en el apartado *SmartSoft Component* dentro de *la vista Palette.*

Al igual que hacíamos antes, para hacerlos formar parte de nuestro diseño, debemos hacer clic con el botón izquierdo de nuestro ratón en el elemento de la paleta, y a continuación clic de nuevo en el mismo botón ahora sobre el modelo. Una vez dichos elementos forman parte del modelo, es posible recolocarlos para una mejor visualización a gusto del usuario manteniendo pulsado el botón izquierdo.

Una vez añadidos estos elementos vamos a modificamos sus nombres en la vista *Properties* de cada elemento (*printClient* y *PrintTask* respectivamente).

Al igual que hacíamos en el caso anterior del servidor, importamos del espacio de trabajo el proyecto *CommSampleObject,* ya que utilizaremos uno de los objetos de comunicación contenidos en este proyecto. En concreto será el *CommSamplePrint*, que también establecimos en el servidor. El proceso es el siguiente:

- En la pestaña Outline: *Botón derecho sobre 'SamplePushNewestServer' / Smart Robotics / Import CommObject Repository.* A continuación buscamos el *'CommSampleObject'.*

El siguiente paso será proceder a configurar el puerto. Para ello accedemos al perfil del mismo: *Properties / Profile*. Fijamos los siguientes valores:

- *serverName : String [1..1] = SampleSendServer*
- *serviceName : String [1..1] = print*
- *commObject : Class [1..1] = CommSamplePrint*

Establecemos por tanto que el servidor de dicho servicio es el componente que hemos creado anteriormente, el *SampleSendServer.* Del mismo modo, especificamos el servicio exacto del que haremos uso, que coincide con el que se definió en el componente servidor. Como ya hemos comentado, el tipo de objeto de comunicación que utiliza el puerto de comunicación del cliente será *CommSamplePrint,* que también coincide con el especificado en el puerto de comunicación del servidor.

El siguiente paso es generar el código asociado al modelo. Para ello, hacemos lo siguiente:

- Botón derecho sobre el proyecto desde la vista *Navigator / Smart Robotics / Run Code Generator*

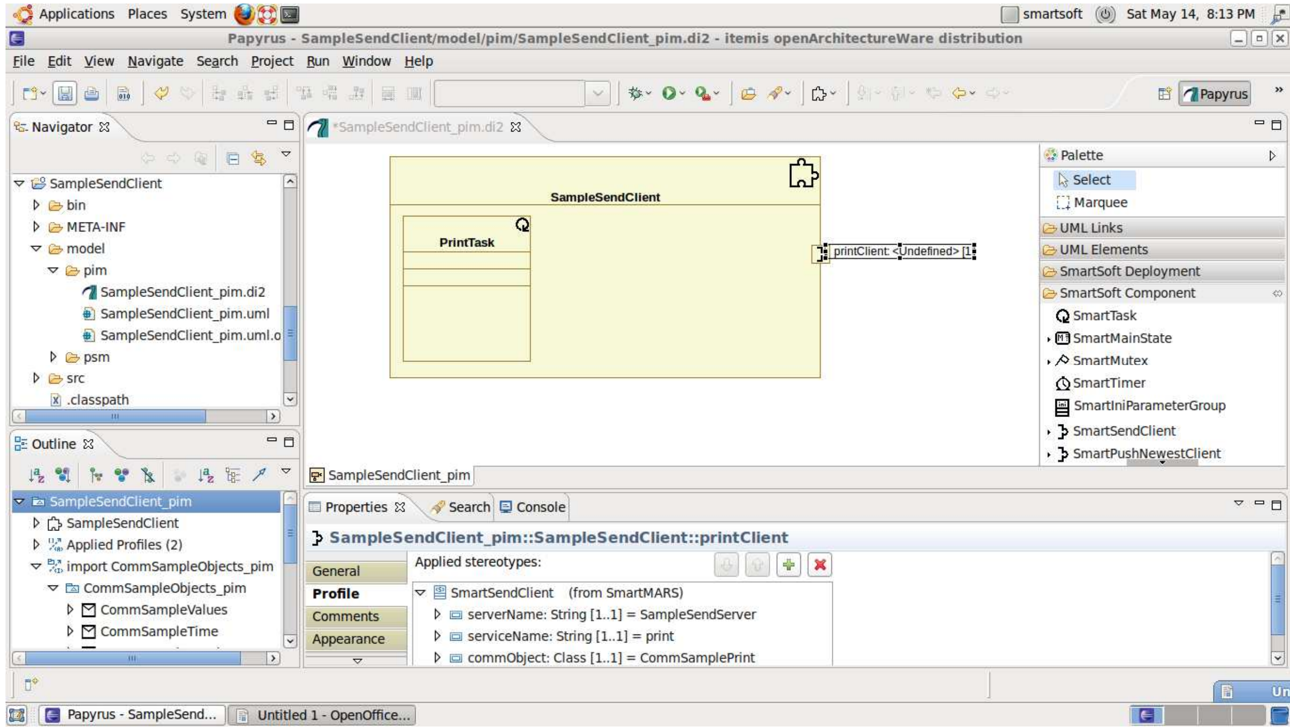


Figura 5.3: Modelado del componente SmartSendClient

Una vez hemos realizado esto, en la carpeta *src* modificamos el fichero *PrintTask.cc.* En este fichero se establece el comportamiento de la tarea *PrintTask*. El código es el siguiente:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
int PrintTask::svc(){
        CommSampleObjects::CommSamplePrint m;
        time_t time_now;
        struct tm *time_p;
        CHS::StatusCode status;

        while(1){
            time_now = time(0);
            time_p    = gmtime(&time_now);
            std::cout << "Soy Print Task y voy a enviar.";
            m.set(time_p->tm_hour,time_p->tm_min,time_p->tm_sec,"Mensaje desde PrintTask: ");
            status = SampleSendClient::instance()->PrintClient->send(m);
        }
        return 0;
}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

La tarea posee un bucle infinito en el que se prepara el objeto de comunicación introduciendo la hora actual (de esta manera seguiremos si se van recibiendo correctamente). A continuación mediante el uso de la función miembro “*send”* propia del cliente del patrón de comunicación “Send”, enviar el objeto al servidor.

Por último realizamos el 'build' sobre el proyecto:

- *Botón derecho sobre el proyecto desde la vista Navigator / Smart Robotics / Build Project*

Construcción del “DeploySampleSend”

Creamos un nuevo proyecto de tipo deployment:

- *File / New / Project → Smart Robotic Project / SmartSoft Deployment →* Fijamos el nombre del componente. ('*DeploySampleSend*").

Abrimos el modelo. Para ello, buscamos en la vista *Navigator* el proyecto recién creado:

- *DeploySampleSend / model / pim / DeploySampleSend.di2*

Lo primero será importar al proyecto los componentes que van a formar parte del deployment. Para importar tanto el *SampleSendClient* como el *SampleSendServer*, hacemos lo siguiente:

- En la pestaña Outline: *Botón derecho sobre 'DeploySampleSend ' / Smart Robotics / Import Component.* A continuación buscamos el *SampleSendClient* y *SampleSendServer.*

Al igual que ya pasaba cuando importábamos el *CommSampleObject*, ahora también será necesario cerrar y volver abrir el proyecto para asegurarse de que todo se ha realizado correctamente.

El siguiente paso, es arrastrar desde la misma pestaña *'Outline'* los componentes al modelo del deployment, así como los puertos que ambos poseen y que vayamos a utilizar (no es necesario arrastrar las tareas). Una vez hecho, pasamos a relacionar los componentes. Para ello hacemos uso de *'Connection'*, que se encuentra en el apartado *SmartSoft Deployment* dentro de la vista *'Palette'.* Lo que hacemos es pinchar sobre el puerto *printClient* del *SampleSendClient*, y llevarlo hasta el *printServer* del *SampleSendServer.*

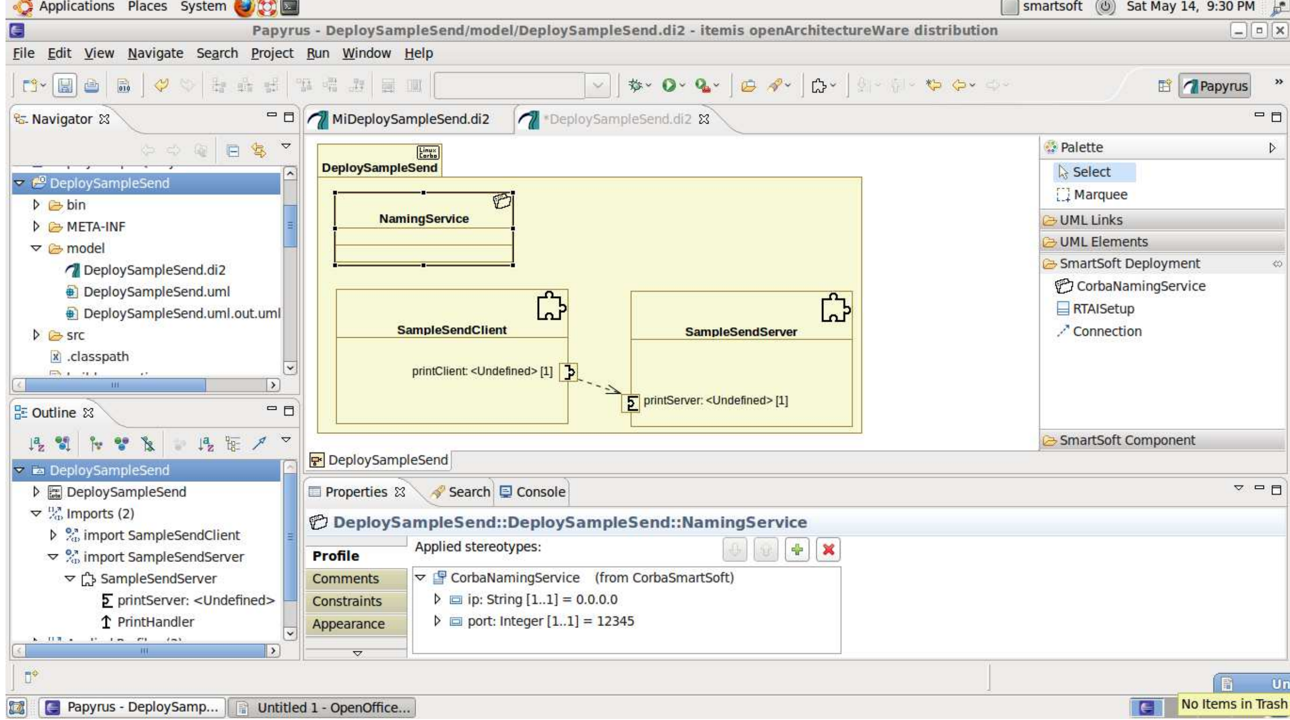


Figura 5.4: Modelado del Deployment DeploySampleSend

A continuación lo que hacemos es introducir el *NamingService* en el deployment. Este elemento también lo encontramos en el apartado *SmartSoft Deployment* dentro de la vista *Palette*. Configuramos su perfil desde el apartado *Profile* de la vista *Properties:*

- *ip : String [1..1] = 0.0.0.0*
- *port : Integer [1..1] = 12345*

Por último realizamos la operación de *Deploy* sobre el proyecto:

- Botón derecho sobre el proyecto desde la vista *Navigator / Smart Robotics / Deploy*

Tras esto, se ha creado en la carpeta *src* correspondiente al proyecto y denominada de igual manera que el deployment, el ejecutable *deploySampleSend.sh.*

Para ejecutarlo escribimos en la consola:

```
>> ./deploySampleSend start
```

Obteniendo los resultados mostrados en la figura 5.5:

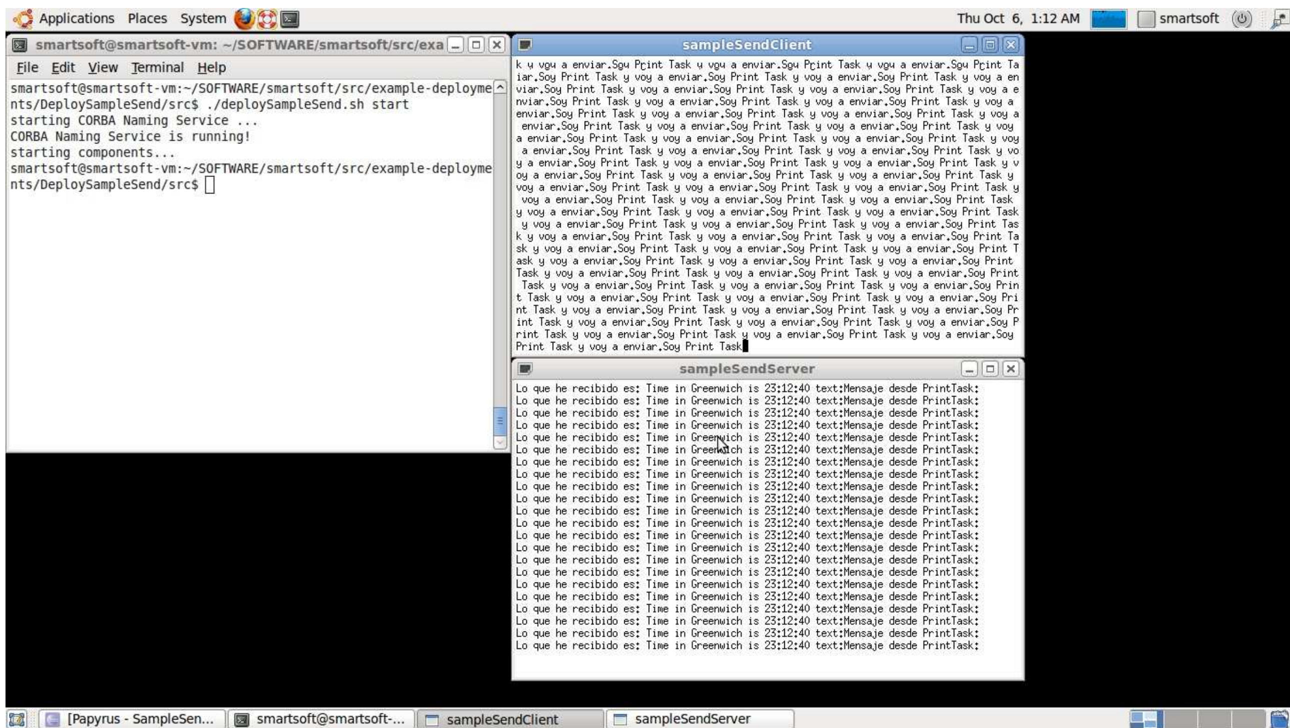


Figura 5.5: Ejecución de DeploySampleSend

Los resultados son los esperados. Para finalizar la ejecución del deployment introducimos el siguiente comando:

```
>> ./deploySampleSend stop
```

5.2. Ejemplo de utilización patrón “Query”

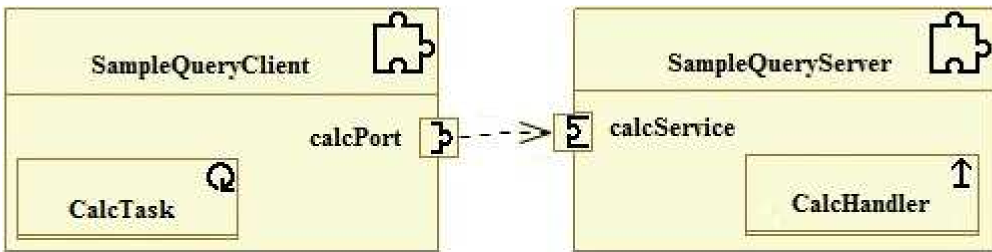


Figura 5.6: Diagrama de funcionamiento ejemplo Query

En este apartado se describe el proceso de construcción de una aplicación sencilla, en la que mostrar el funcionamiento del patrón de comunicación “Query”. Se construyen dos componentes: un cliente y un servidor. El cliente prepara un objeto de comunicación que contiene una lista de enteros, lo envía al servidor para que éste realice un procesamiento sobre ellos (el cual es una suma de todo el conjunto de números), y por último devuelve al cliente el resultado obtenido. Ya que el patrón Query permite el uso de diferentes objetos de comunicación para la solicitud y la respuesta, vamos a utilizar esta opción.

El componente servidor se denomina *SampleQueryServer.* Este componente tiene un puerto del tipo *SmartQueryServer* a través del cual recibe el objeto de comunicación con el que trabaja. El objeto de comunicación recibido es del tipo *CommSampleValues*, mientras que el devuelve un *CommSampleResult.* Ambos se encuentran en el repositorio *CommSampleObjetcs*. Por otro lado, el servidor posee un handler del tipo *SmartQueryHandler* que se lanza cada vez que recibe un objeto de comunicación a través del puerto, y donde realizar su procesamiento. La función con la que devolver el resultado al cliente es *anwer.*

Por otra parte, tenemos el componente *SampleQueryClient* con un puerto del tipo *SmartQueryClient* y una tarea *SmartTask*. La tarea se encarga por un lado de preparar la lista de enteros que contiene el objeto *CommSampleValues*, y tras enviarla al servidor mediante la función *query* (método bloqueante), recibir la respuesta en forma de *CommSampleResult* e imprimirlo por pantalla. Para que se realice un mejor seguimiento de la ejecución, y el cliente no enviará otra petición hasta un segundo después. Para ello se utiliza la variable *queryDelayMs* definida en el modelo del cliente.

Construidos los dos componentes, se realiza un deployment denominado *DeploySampleQuery* en el que relacionar ambos componentes. Dividimos por tanto el diseño en tres secciones:

- Construcción del componente *SampleQueryServer.*
- Construcción del componente *SampleQueryClient.*
- Construcción y ejecución del deployment *DeploySampleQuery.*

Construcción del componente servidor “SampleQueryServer”

Creamos un nuevo proyecto de tipo Componente:

- *File / New / Project* → *Smart Robotic Project / SmartSoft Component* → Fijamos el nombre del componente: '*SampleQueryServer*'

Abrimos el modelo. Para ello, buscamos en la pestaña *Navigator* el proyecto recién creado:

- *SampleQueryServer / model / pim / SampleQueryServer_pim.di2*

Este componente está formado por el puerto de comunicación del servidor *SmartQueryServer* y un handler asociado del tipo *SmartQueryHandler.* Estos dos elementos los encontramos en el apartado *SmartSoft Componen*t dentro de la vista *Palette*. Para hacerlos formar parte de nuestro diseño, para cada uno de ellos debemos hacer clic con el botón izquierdo de nuestro ratón en el elemento de la paleta, y a continuación clic

de nuevo ahora sobre el modelo. Una vez dichos elementos forman parte del modelo, es posible recolocarlos para una mejor visualización a gusto del usuario manteniendo pulsado el botón izquierdo.

Una vez añadidos estos elementos vamos a modificar sus nombres. Esto lo podemos hacer en las *Properties / General* de cada elemento. Los renombramos como: *calcService* y *CalcHandler* respectivamente (se recomienda que los puertos tengan la primera letra en minúscula y de esta manera evitaremos un “Warning” en el proceso de generación de código).

A continuación vamos a importar al espacio de trabajo el proyecto *CommSampleObject*. En este repositorio se incluyen los objetos de comunicación *CommSampleValues* y *CommSampleResult.*

Lo hacemos de la siguiente manera:

- En la pestaña Outline: *Botón derecho sobre 'SampleQueryServer' / Smart Robotics / Import CommObject Repository.* A continuación buscamos el *'CommSampleObject'. C*errar y volver abrirlo para asegurarnos de que se ha realizado correctamente.

El siguiente paso será proceder a configurar el puerto. Para ello accedemos al perfil del mismo: *Properties / Profile.* Fijamos los siguientes valores:

– *smartQueryHandler: smartQueryHandler [1..1] = CalcHandler*
– *commRequestObject : Class [1..1] = CommSampleValues*
– *commAnswerObject: Class [1..1] = CommSampleResult*
– *serviceName : String [1..1] = calc*

Se vincula el puerto de comunicación con el handler, así definir el tipo de objeto de comunicación para la solicitud y respuesta. Por último establecemos el identificador que tendrá el servicio.

El último paso que realizamos con el modelo es generar el código asociado al diseño:

- *Botón derecho sobre el proyecto en la vista Navigator / Smart Robotics / Run Code Generator*

Tras haber generado el código asociado, en la carpeta *src* del proyecto buscamos el fichero *CalcHandler.cc*. En este fichero se establece el comportamiento del handler asociado al puerto *SampleQueryServer.*

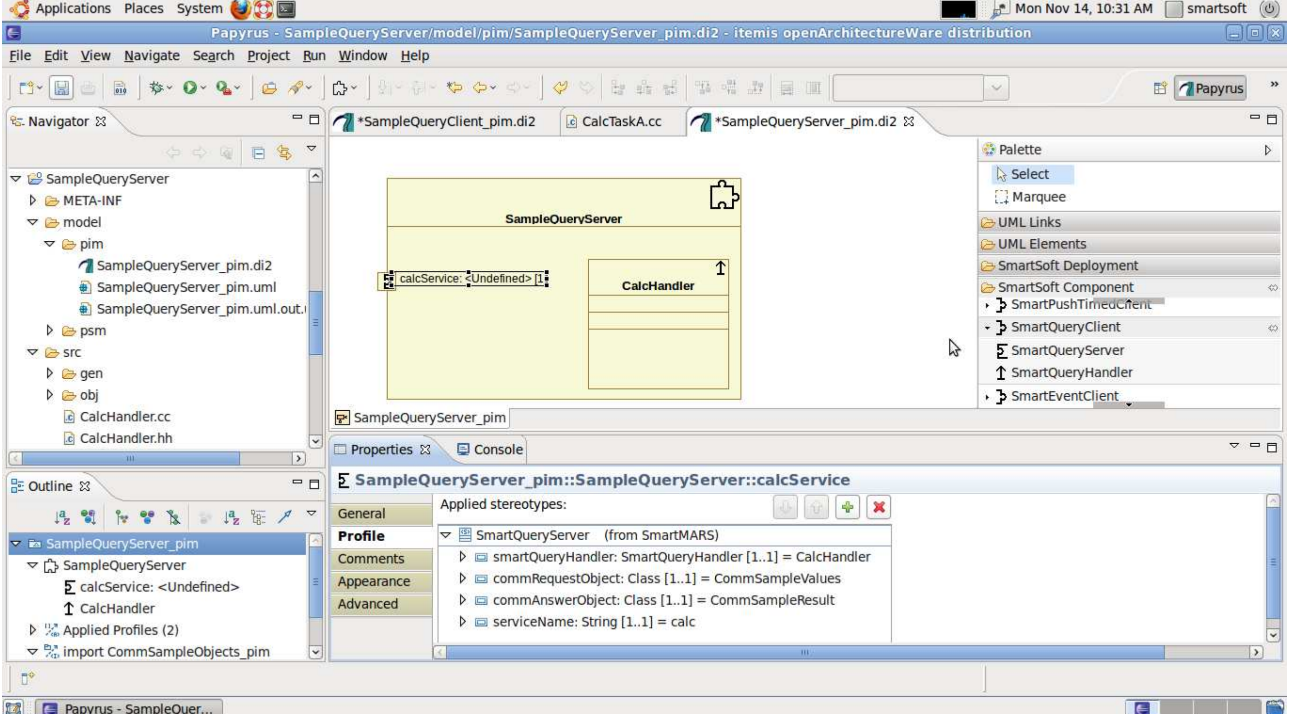


Figura 5.7: Modelado del componente SampleQueryServer

%%
```
void CalcHandler::handleQuery(CHS::QueryServer<CommSampleObjects::CommSampleValues,
            CommSampleObjects::CommSampleResult> & server, const CHS::QueryId id,
            const CommSampleObjects::CommSampleValues & request) throw (){

      CommSampleObjects::CommSampleResult answer;
      std::list<int> list;
      int result;

      std::cout << "calc service " << id << std::endl;

       request.get(list);
       result = 0;
       for (std::list<int>::iterator i=list.begin();i!=list.end();++i) {
            result += *i;
      }
      answer.set(result);
      std::cout << "calc service " << id << " sent answer " << result << std::endl;

      server.answer(id, answer);
}
```
%%

La ejecución del handler comienza imprimiendo “el número de servicio” que va a tratar. Dicho valor el cliente lo envía automáticamente al realizar la llamada al método *query*. A continuación suma todos los valores de la lista que contiene el objeto *CommSampleValues*. Tras su cálculo, muestra el resultado por pantalla, lo introduce en el objeto *CommSampleResult* y lo envía al cliente mediante la función *answer.*

Ya solo nos queda realizar el 'Build' sobre el proyecto:

- *Botón derecho sobre el proyecto desde la vista Navigator / Smart Robotics / Build Project*

Construcción del componente cliente “SampleQueryClient”

Creamos un proyecto de tipo componente:

- *File / New / Project → Smart Robotic Project / SmartSoft Component →* Fijamos el nombre del componente. ('*SampleQueryClient*')

Abrimos el modelo:

- *Desde la vista Navigator / SampleQueryClient / model / pim / SampleQueryClient_pim.di2*

Este componente está compuesto por un puerto *SmartQueryClient* y una tarea *SmartTask*. Una vez añadidos modificamos sus nombres en la vista *Properties* de cada elemento (*calcPort* y *CalcTask* respectivamente).

Al igual que hacíamos en el caso anterior del servidor, importamos del espacio de trabajo el proyecto *CommSampleObject,* ya que los dos objetos de comunicación utilizados se encuentran en dicho repositorio. El proceso es el siguiente:

- En la pestaña Outline: *Botón derecho sobre 'SampleQueryServer' / Smart Robotics / Import CommObject Repository.* A continuación buscamos el *'CommSampleObject'.*

El siguiente paso será proceder a configurar el puerto. Para ello accedemos al perfil del mismo: *Properties / Profile.* Fijamos los siguientes valores:

- *serverName : String [1..1] = SampleQueryServer*
- *serviceName : String [1..1] = calc*
- *commRequestObject : Class [1..1] = CommSampleValues*
- *commAnswerObject : Class [1..1] = CommSampleResult*

Establecemos por tanto que el servidor de dicho servicio es el componente que hemos creado anteriormente, el *SampleQueryServer.* Del mismo modo, especificamos el servicio exacto del que haremos uso, que coincide con el que se definió en el componente servidor. Como ya hemos comentado, el tipo de objeto de comunicación que utiliza el puerto de comunicación del cliente para la solicitud será *commSampleValues* y recibe como respuesta un *commSampleResult.*

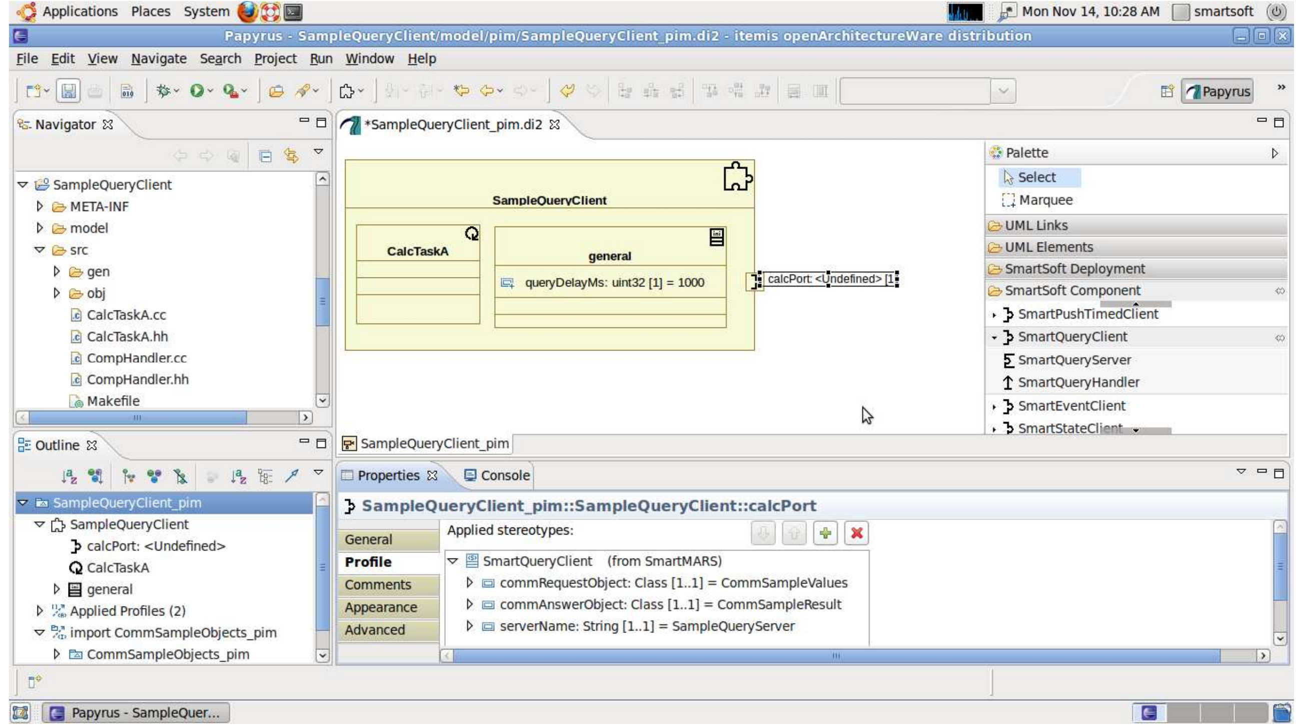


Figura 5.8: Modelado del componente SmartQueryClient

El siguiente paso es generar el código asociado al modelo. Para ello, hacemos lo siguiente:

- Botón derecho sobre el proyecto desde la vista *Navigator / Smart Robotics / Run Code Generator*

Una vez hemos realizado esto, en la carpeta *src* modificamos el fichero *CalcTask.cc*. En este fichero se establece el comportamiento de la tarea *CalcTask*. El código es el siguiente:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
int CalcTaskA::svc(){

        COMP->calcPort->connect( COMP->ini.calcPort.serverName, COMP >ini.calcPort.serviceName);

        int i=0;
        std::list<int> list;

        CommSampleObjects::CommSampleValues request;
        CommSampleObjects::CommSampleResult result;

        CHS::StatusCode status;

        while (1){
                list.clear();
                list.push_back(2);
                list.push_back(3);
                list.push_back(4);

                request.set(list);
                std::cout << "query " << i++ << std::endl;

                status = COMP->calcPort->query(request, result);

                std::cout << "query (status): " << CHS::StatusCodeConversion(status) << " result ";
                result.print();
                usleep( COMP->ini.general.queryDelayMs * 1000 );
        }
        return 0;
}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

La tarea posee un bucle infinito en el que se prepara la lista de números del objeto *CommSampleValues*, para a continuación enviarlo al servidor mediante la función *query* (llamada bloqueante). Notar que al servidor se

le envía tanto el objeto de solicitud como el de respuesta. Una vez obtenido el resultado, se imprime por pantalla. Por último, para mejor seguimiento de la ejecución, fijamos que el cliente realice consultas al servidor cada segundo. Utilizamos el valor de la variable *queryDelayMs* definida en el modelo como parámetro de la función *usleep.*

Por último realizamos el 'build' sobre el proyecto:

- *Botón derecho sobre el proyecto desde la vista Navigator / Smart Robotics / Build Project*

Construcción del “DeploySampleQuery”

Creamos un nuevo proyecto de tipo deployment:

- *File / New / Project → Smart Robotic Project / SmartSoft Deployment →* Fijamos el nombre del componente. ('*DeploySampleQuery*").

Abrimos el modelo. Para ello, buscamos en la vista *Navigator* el proyecto recién creado:

- *DeploySampleQuery / model / pim / DeploySampleQuery.di2*

Lo primero será importar al proyecto los componentes que van a formar parte del deployment. Para importar tanto el *SampleQueryClient* como el *SampleQueryServer*, hacemos lo siguiente:

- En la pestaña Outline: *Botón derecho sobre 'DeploySampleSend ' / Smart Robotics / Import Component.* A continuación buscamos el *SampleQueryClient* y *SampleQueryServer.*

Al igual que ya pasaba cuando importábamos el *CommSampleObject*, ahora también será necesario cerrar y volver abrir el proyecto para asegurarse de que todo se ha realizado correctamente.

El siguiente paso, es arrastrar desde la misma pestaña *'Outline'* los componentes al modelo del deployment, así como los puertos que ambos poseen y que vayamos a utilizar (no es necesario arrastrar las tareas y handlers).

Una vez hecho, pasamos a relacionar los componentes. Para ello hacemos uso de *'Connection'*, que se encuentra en el apartado *SmartSoft Deployment* dentro de la vista *'Palette'*. Lo que hacemos es pinchar sobre el puerto *printClient* del *SampleSendClient*, y llevarlo hasta el *printServer* del *SampleSendServer.*

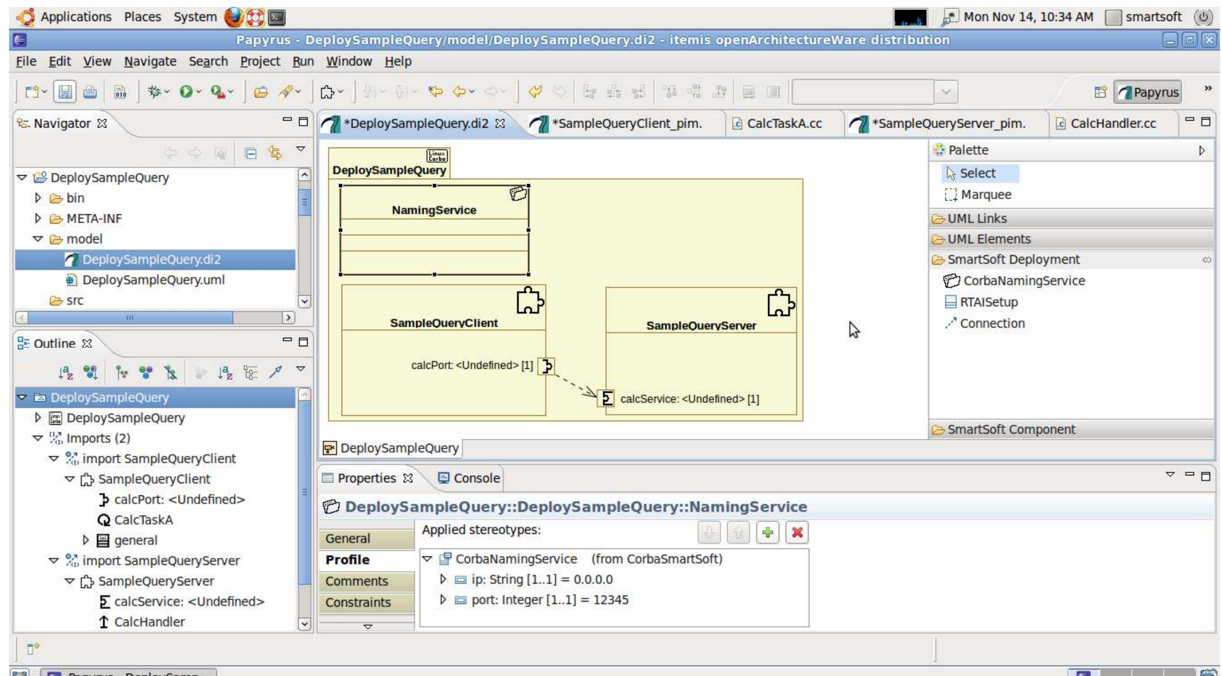


Figura 5.9: Modelado del Deployment DeploySampleQuery

A continuación se introduce el *NamingService* en el deployment. Este elemento también lo encontramos en el apartado *SmartSoft Deployment* dentro de la vista *Palette*. Lo configuramos desde el apartado *Profile* de la vista *Properties:*

- *ip : String [1..1] = 0.0.0.0*
- *port : Integer [1..1] = 12345*

Por último realizamos la operación de *Deploy* sobre el proyecto:

- Botón derecho sobre el proyecto desde la vista *Navigator / Smart Robotics / Deploy*

Tras esto, se ha creado en la carpeta *src* correspondiente al proyecto y denominada de igual manera que el deployment, el ejecutable *deploySampleQuery.sh.*

Para ejecutarlo escribimos en la consola:

```
>> ./deploySampleQuery start
```

Obteniendo los resultados mostrados en la figura 5.10:

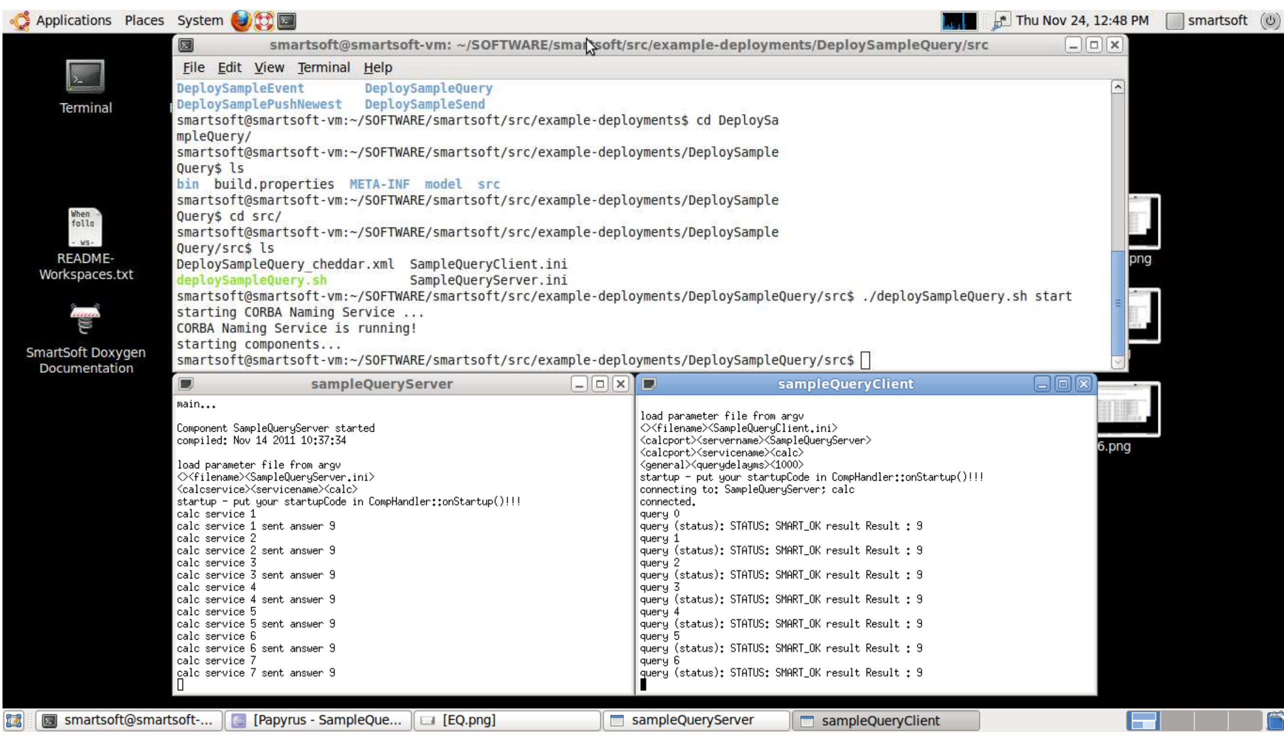


Figura 5.10: Ejecución de DeploySampleQuery

Los resultados son los esperados, obteniendo como resultado *"2+3+4 = 9"*. También se observa cómo se van numerando los *"calc service"* en el servidor, sin la necesidad de que este lleve un conteo propio, sino que el *id* es incluido en el upcall del *CalcHandler.*

Para finalizar la ejecución del deployment introducimos el siguiente comando:

```
>> ./deploySampleQuery stop
```

5.3. Ejemplo de utilización patrón “PushNewest”

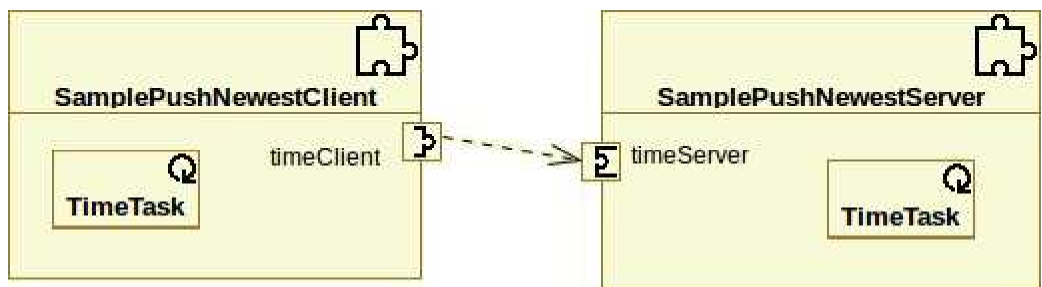


Figura 5.11: Diagrama de funcionamiento ejemplo PushNewest

En este apartado se describe el proceso de construcción de una aplicación sencilla, en la que mostrar el funcionamiento del patrón de comunicación “PushNewest”. Se construyen dos componentes: un cliente y un servidor. En este ejemplo, el servidor obtiene la hora y la envía al cliente suscrito repetidamente. La tasa de refresco es de 1 segundo, y el cliente quedará bloqueado entre actualizaciones.

Creamos un componente denominado *SamplePushNewestServer.* Este componente tiene un puerto del tipo *SmartPushNewestServer* a través del cual distribuye el objeto de comunicación (hora, minutos y segundos) a los distintos clientes suscritos mediante la función *put*. Además posee una tarea *SmartTask* donde prepara el objeto de comunicación *CommSampleTime* utilizando la función *set* del objeto. Por otra parte, tenemos el componente *SamplePushNewestClient* con un puerto del tipo *SmartPushNewestClient* y una tarea *SmartTask*.

La tarea se encarga de recibir el objeto *CommSampleTime* mediante la función *getUpdateWait*, quedando bloqueado hasta conseguirlo. A continuación ayudada de la función *get*, obtiene los distintos campos de los que se compone el objeto recibido e imprime la hora.

Construidos los dos componentes, se realiza un deployment denominado *DeploySamplePushNewest* en el que relacionar ambos componentes. Dividimos el ejemplo en tres secciones:

- Construcción del componente *SamplePushNewestServer.*
- Construcción del componente *SamplePushNewestClient.*
- Construcción y ejecución del deployment *DeploySamplePushNewest.*

Construcción del componente servidor “SamplePushNewestServer”

Creamos un nuevo proyecto del tipo Componente:

- *File / New / Project* → *Smart Robotic Project / SmartSoft Component* → Fijamos el nombre del componente: '*SamplePushNewestServer*'

Abrimos el modelo. Para ello, buscamos en la pestaña *Navigator* el proyecto recién creado:

- *SamplePushNewestServer / model / pim / SamplePushNewestServer_pim.di2*

Este componente está compuesto por un puerto del tipo *SmartPushNewestServer* y una tarea *SmartTask.* Estos 2 elementos los encontramos en el apartado *SmartSoft Componen*t dentro de *Palette*.

Para hacerlos formar parte de nuestro diseño, para cada uno de ellos debemos hacer clic con el botón izquierdo de nuestro ratón en el elemento de la paleta, y a continuación hacer clic de nuevo ahora sobre el modelo. Una vez dichos elementos forman parte del modelo, es posible recolocarlos para una mejor visualización a gusto del usuario manteniendo pulsado el botón izquierdo.

Una vez añadidos estos elementos vamos a modificar sus nombres. Esto lo podemos hacer en las *Properties / General* de cada elemento. Los renombramos como: *timeServer* y *TimeTask* respectivamente (se recomienda que los puertos tengan la primera letra en minúscula y de esta manera evitaremos un “Warning” en el proceso de generación de código a partir del modelo).

A continuación vamos a importar al espacio de trabajo el proyecto *CommSampleObject*. En este, se incluyen objetos de comunicación básicos como es el *CommSampleTime.*

Es posible importar únicamente el objeto que vayamos a utilizar en nuestro diseño en lugar de hacerlo con el proyecto o repositorio completo. Sin embargo, debido a que han surgido algunos errores cuando se han importado objetos sueltos, y a la comodidad que supone importar el proyecto completo, decidimos realizarlo de este modo

Lo hacemos de la siguiente manera:

- En la pestaña Outline: *Botón derecho sobre 'SamplePushNewestServer' / Smart Robotics / Import CommObject Repository.* A continuación buscamos el *'CommSampleObject'.*

Para comprobar que el proceso se ha realizado correctamente, debido a un problema de SmartSoft, será necesario cerrar el proyecto y volver abrirlo.

El siguiente paso será proceder a configurar el puerto. Para ello accedemos al perfil del mismo: *Properties / Profile.* Fijamos los siguientes valores:

– *commObject : Class [1..1] = CommSampleTime*
– *serviceName : String [1..1] = time*

En este caso, al contrario que pasaba con el patrón de comunicación Send donde el servidor tenía un handler, ahora el servidor posee una tarea *SmartTask.* Podemos ver como en el *Profile* del puerto no se especifica nada con respecto a la tarea. Ambos elementos se vincularán en el fichero *TimeTask.cc* como veremos más adelante. También será necesario establecer el nombre que tendrá el servicio entre componentes.

El siguiente paso es generar el código asociado al modelo. Para ello, hacemos lo siguiente:

- *Botón derecho sobre el proyecto / Smart Robotics / Run Code Generator*

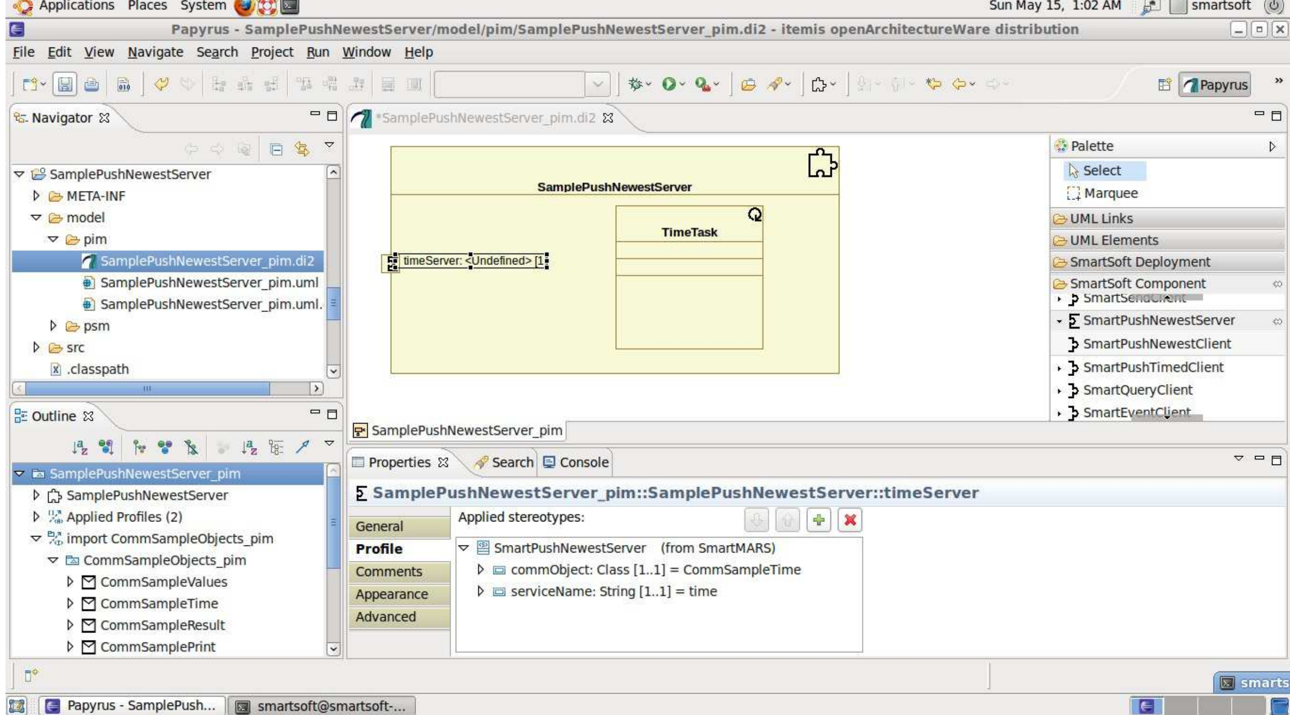


Figura 5.12: Modelado del componente SmartPushNewestServer

Una vez hemos realizado esto, en la carpeta *src* de nuestro proyecto aparecen distintos ficheros. En este caso, nosotros estamos interesados en el fichero *TimeTask.cc*, donde se detalla el comportamiento de la tarea. El código es el siguiente:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
int TimeTask::svc(){
        time_t time_now;
        struct tm *time_p;
        CHS::StatusCode status;
        int i=0;

        CommSampleObjects::CommSampleTime a;

        while(1){
                time_now = time(0);
                time_p   = gmtime(&time_now);
                a.set(time_p->tm_hour,time_p->tm_min,time_p->tm_sec);
                status = COMP->timeServer->put(a);
                std::cout << "thread A <push new data> " << i++ << " status: " <<
CHS::StatusCodeConversion(status) << " ";a.print();
                sleep(1);
          }
        return 0;
}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

El bucle infinito que posee la tarea se limita a obtener la hora actual, preparar el objeto de comunicación con la función *set*, y enviársela al cliente gracias a la función *put*. Para seguir la ejecución de una manera más clara, el servidor imprime lo que envía. Por último paramos la ejecución de la tarea durante un segundo mediante la función *sleep*.

Ya solo nos queda realizar el 'build' sobre el proyecto:

- *Botón derecho sobre el proyecto / Smart Robotics / Build Project*

Construcción del componente cliente “SamplePushNewestClient”

Creamos un proyecto nuevo:

- *File / New / Project → Smart Robotic Project / SmartSoft Component* → Fijamos el nombre del componente '*SamplePushNewestClient'*

Abrimos el modelo:

- *SamplePushNewestClient / model / pim / SamplePushNewestClient_pim.di2*

Este componente está compuesto por un puerto *SmartPushNewestClient* y una tarea *SmartTask.* Ambos elementos los encontramos en el apartado *SmartSoft Component* dentro de *Palette.*

Al igual que hacíamos antes, para hacerlos formar parte de nuestro diseño, debemos hacer clic con el botón izquierdo de nuestro ratón en el elemento de la paleta, y a continuación hacer clic de nuevo ahora sobre el modelo. Una vez dichos elementos forman parte del modelo, es posible recolocarlos para una mejor visualización a gusto del usuario manteniendo pulsado el botón izquierdo. Una vez añadidos estos elementos vamos a modificamos sus nombres en la vista *Properties* de cada elemento (*timeClient* y *TimeTask* respectivamente).

Importamos del espacio de trabajo el proyecto *CommSampleObject,* ya que utilizaremos uno de los objetos de comunicación contenidos en este proyecto. En concreto será el *CommSampleTime*, que también establecimos en el servidor.

El proceso es el siguiente:

- En la pestaña Outline: Botón derecho sobre *'SamplePushNewestServer' / Smart Robotics / Import CommObject Repository.* A continuación buscamos el *'CommSampleObject'.*

El siguiente paso será proceder a configurar el puerto. Para ello accedemos al perfil del mismo: *Properties / Profile*. Fijamos los siguientes valores:

- *serverName : String [1..1] = SamplePushNewestServer*
- *serviceName : String [1..1] = time*
- *commObject : Class [1..1] = CommSampleTime*

Establecemos por tanto que el servidor de dicho servicio es el componente que hemos creado anteriormente, el *SamplePushNewestServer.* Del mismo modo, especificamos el servicio exacto del que haremos uso, que coincide con el que se definió en el componente servidor. El tipo de objeto de comunicación utilizado el será *CommSampleTime,* que también coincide con el especificado en el puerto de comunicación del servidor.

El siguiente paso es generar el código asociado al modelo. Para ello, hacemos lo siguiente:

- *Botón derecho sobre el proyecto / Smart Robotics / Run Code Generator*

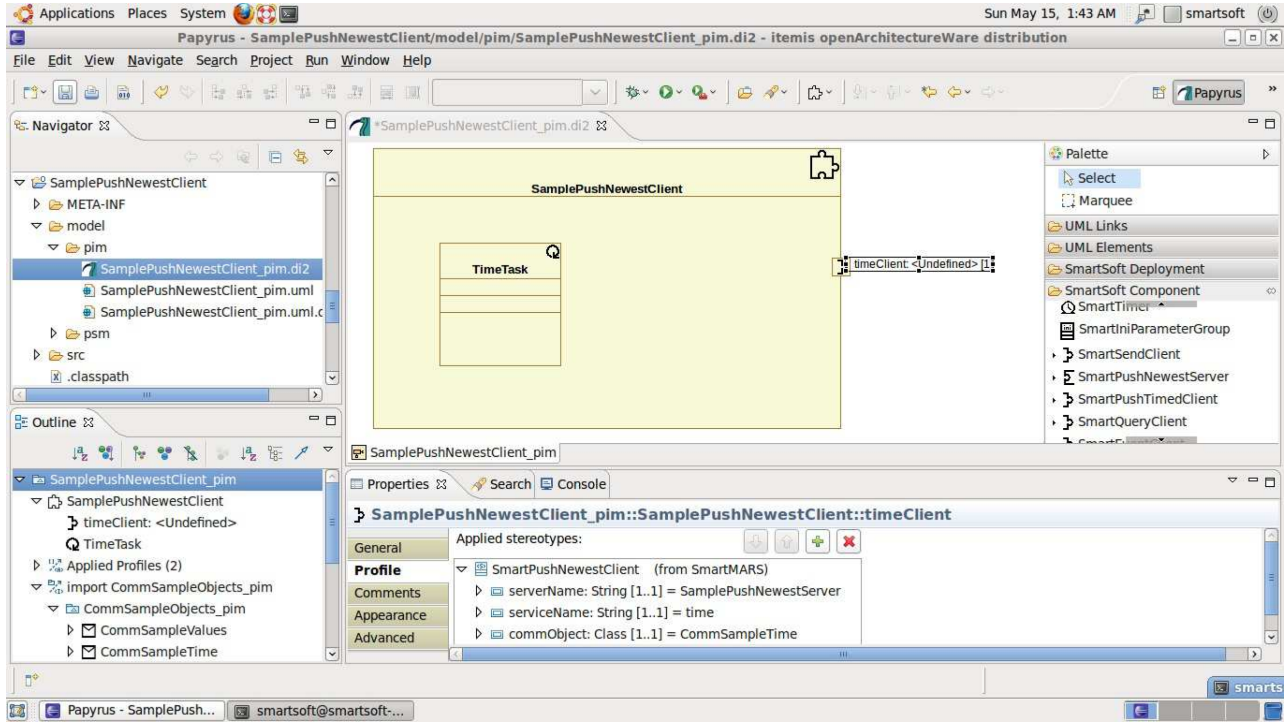


Figura 5.13: Modelado del componente SmartPushNewestClient

Una vez hemos realizado esto, en la carpeta *src* de nuestro proyecto aparecen distintos ficheros.
En este caso, nosotros estamos interesados en el fichero *TimeTask.cc*. En este fichero se detalla el comportamiento de la tarea. El código es el siguiente:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
int TimeTask::svc(){
        CommSampleObjects::CommSampleTime a;
        CHS::StatusCode status;
        int h, m, s;
        COMP->timeClient->subscribe();

        while(1){
                status = COMP->timeClient->getUpdateWait(a);
                if (status == CHS::SMART_OK){
                        a.get(h, m, s);
                        std::cout << "hora: " << h << " minuto: " << m << " segundo: " << s <<
std::endl;
                }
        }
        return 0;
}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Tras suscribirse al servicio, el cliente entra en un bucle infinito donde se dedica a esperar los objetos enviados por parte del servidor, quedando bloqueado entre envíos.

Por último realizamos el 'build' sobre el proyecto:

- *Botón derecho sobre el proyecto / Smart Robotics / Build Project*

Construcción del "DeploySamplePushNewest"

Creamos un nuevo proyecto de tipo deployment:

- *File / New / Project* → *Smart Robotic Project / SmartSoft Deployment* → Fijamos el nombre del componente. ('*DeploySamplePushNewest*").

Abrimos el modelo. Para ello, buscamos en la vista *Navigator* el proyecto recién creado:

- *DeploySamplePushNewest / model / pim / DeploySamplePushNewest.di2*

Lo primero será importar al proyecto los componentes que van a formar parte del deployment. Para importar tanto el *SamplePushNewestClient* como *SamplePushNewestServer*, hacemos lo siguiente:

- En la pestaña Outline: Botón derecho sobre *'DeploySampleSend ' / Smart Robotics / Import Component*. A continuación buscamos el *SamplePushNewestClient* y *SamplePushNewestServer.*

Al igual que ya pasaba cuando importábamos el *CommSampleObject*, ahora también será necesario cerrar y volver abrir el proyecto para asegurarse de que todo se ha realizado correctamente.

El siguiente paso, es arrastrar desde la misma pestaña *Outline*, los componentes al modelo del deployment, así como los puertos que ambos poseen.

Una vez hecho, pasamos a relacionar los componentes. Para ello hacemos uso de *'Connection'*, que se encuentra en el apartado *SmartSoft Deployment* dentro de la vista *Palette*. Lo que hacemos es clicar sobre el puerto *timeClient* del *SamplePushNewestClient*, y clicar de nuevo en el *timeServer* del *SamplePushNewestServer.*

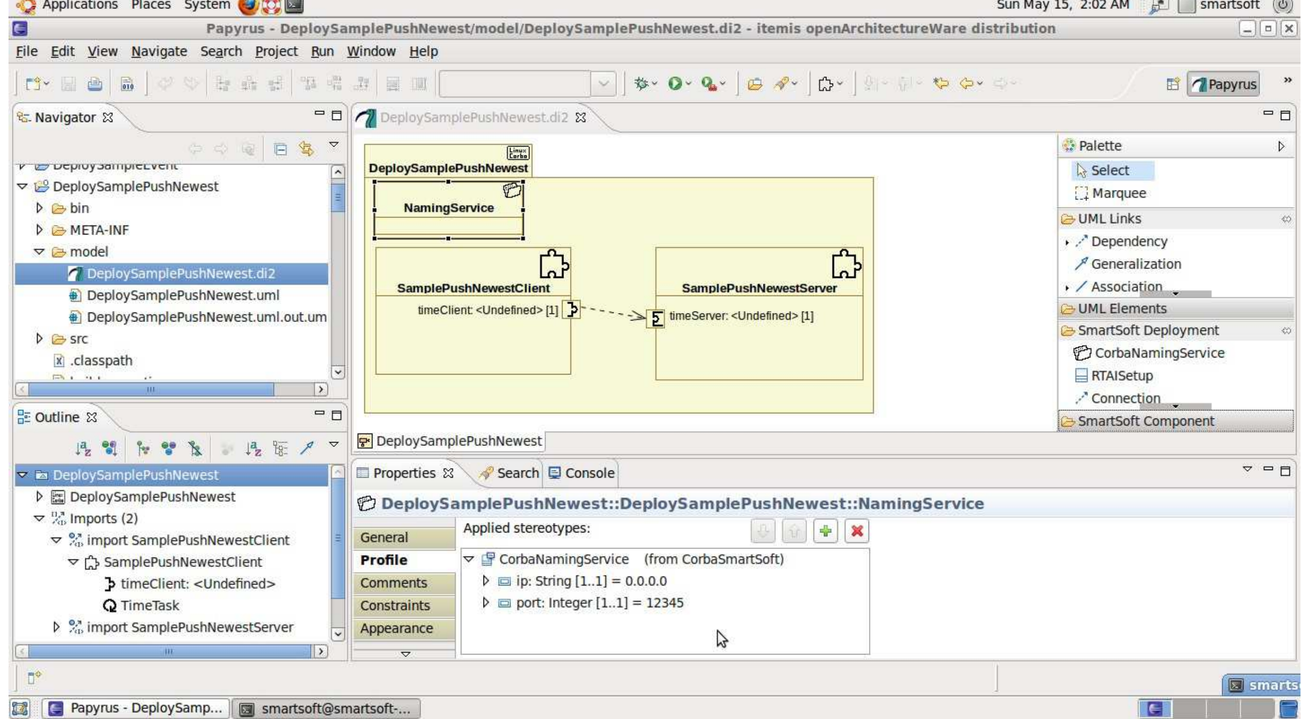


Figura 5.14: Modelado del Deployment DeploySamplePushNewest

A continuación lo que hacemos es introducir el *NamingService* en el deployment. Este elemento también lo encontramos en el apartado *SmartSoft Deployment* dentro de la vista *Palette.*

Modificamos el profile del NamingService (*Properties / Profile*):

- *ip : String [1..1] = 0.0.0.0*
- *port : Integer [1..1] = 12345*

Por último realizamos el Deploy del proyecto:

- *Botón derecho sobre el proyecto / Smart Robotics / Deploy*

Tras esto, se ha creado en la carpeta *src* correspondiente al proyecto y denominada de igual manera que el deployment, el ejecutable *deploySamplePushNewest.sh.*

Para ejecutarlo escribimos en la consola:

```
>> ./deploySamplePushNewest start
```

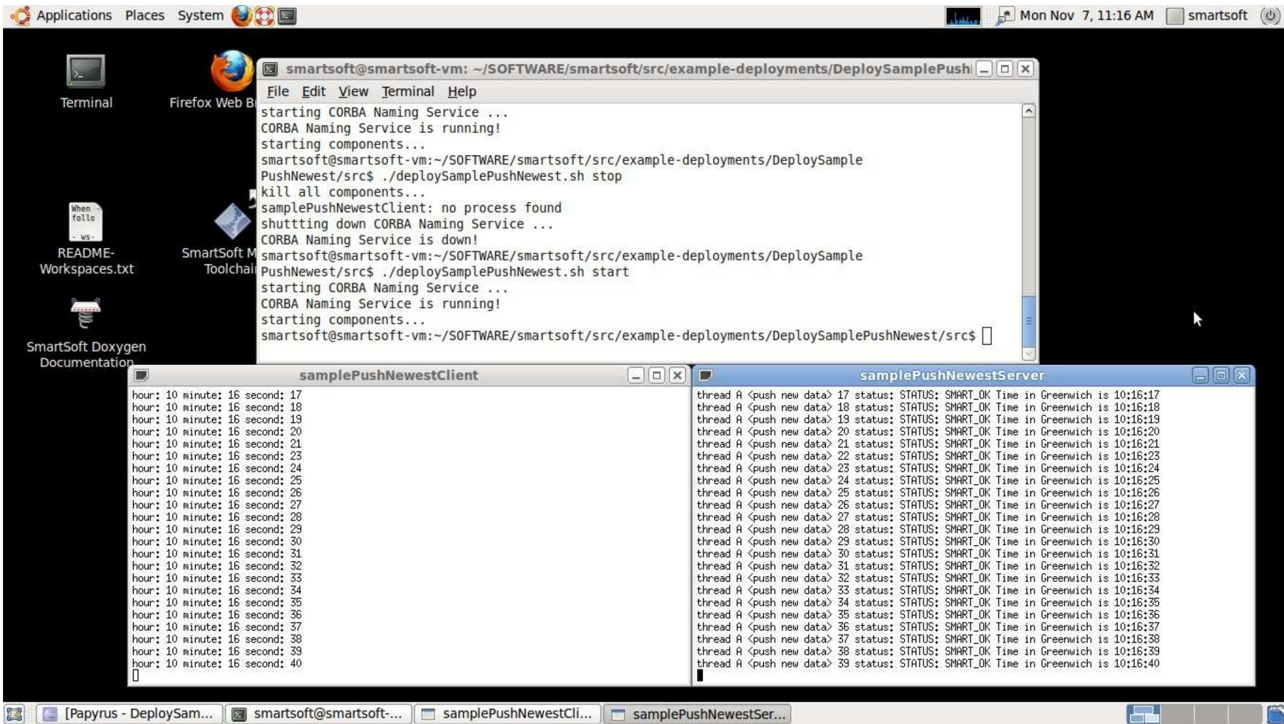


Figura 5.15: Ejecución de DeploySamplePushNewest

Los resultados son los esperados. Para finalizar la ejecución del deployment introducimos el siguiente comando:

```
>> ./deploySamplePushNewest stop
```

5.4. Ejemplo de utilización patrón “PushTimed”

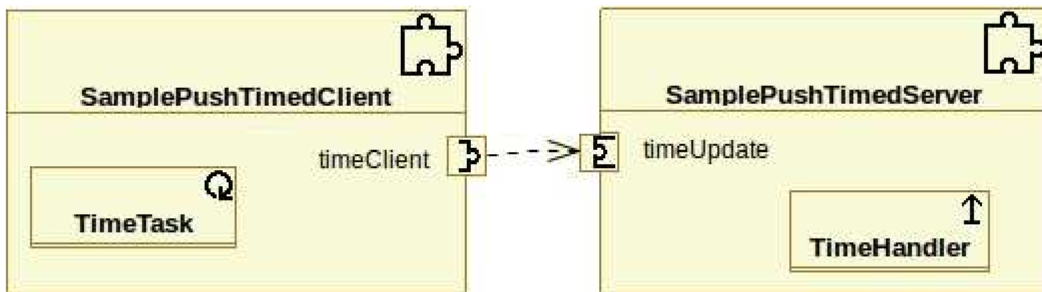


Figura 5.16: Diagrama de funcionamiento ejemplo PushTimed

En este apartado se describe el proceso de construcción de una aplicación sencilla, en la que mostrar el funcionamiento del patrón de comunicación “PushTimed” siguiendo la idea del ejemplo anterior. De nuevo se construyen dos componentes: un cliente y un servidor. El servidor obtiene la hora y la envía al cliente ininterrumpidamente con un periodo de 500 milisegundos. El cliente atenderá los objetos de comunicación con un intervalo de tres recepciones. Esto quiere decir que una vez se ha aceptado un objeto, para aceptar el siguiente deben descartarse antes dos objetos. De esta manera el cliente atenderá un objeto de comunicación entrante cada: *500ms * 3 = 1,5 seg.*

Creamos un componente denominado *SamplePushTimedServer.* Este componente tiene un puerto del tipo *SmartPushTimedServer* a través del cual distribuye el objeto de comunicación (hora, minutos y segundos). Además posee un handler *TimeHandler* del tipo SmartPushTimedHandler. El handler se lanzará cada 500ms como así establecemos en el puerto vinculado *SmartPushTimedServer.* Su ejecución se encarga de preparar el objeto de comunicación *CommSampleTime* utilizando la función *set,* y a continuación a distribuirlo a los distintos clientes suscritos mediante la función *put.*

Por otra parte, tenemos el componente *SamplePushTimeClientClient* con un puerto del tipo *SmartPushTimedClient* y una tarea *SmartTask*. La tarea se encarga de recibir el objeto *CommSampleTime* mediante la función *getUpdateWait*, quedando bloqueado hasta conseguirlo. A continuación haciendo uso de la función *print* definida en la interfaz de usuario del objeto de comunicación, se imprime la hora recibida.

Construidos los dos componentes, se realiza un deployment denominado *DeploySamplePushTimed* en el que relacionar ambos componentes. Dividimos el ejemplo en tres secciones:

- Construcción del componente *SamplePushTimedServer.*
- Construcción del componente *SamplePushTimedClient.*
- Construcción y ejecución del deployment *DeploySampleTimedNewest.*

Construcción del componente servidor “SamplePushTimedServer”

Creamos un nuevo proyecto del tipo Componente:

- *File / New / Project → Smart Robotic Project / SmartSoft Component* → Fijamos el nombre del componente. (p.ej '*SamplePushTimedServer*')

Vamos a abrir el modelo. Para ello, buscamos en la pestaña *Navigator* el proyecto recién creado:

- *SamplePushTimedServer / model / pim / SamplePushTimedServer_pim.di2*

Este componente está compuesto por un puerto del tipo *SmartPushTimedServer* y el handler asociado a este patrón *SmartPushTimeHandler*. Estos dos elementos los encontramos en el apartado *SmartSoft Component* dentro de *Palette*.

Para hacerlos formar parte de nuestro diseño, para cada uno de ellos debemos hacer clic con el botón izquierdo de nuestro ratón en el elemento de la paleta, y a continuación clic de nuevo ahora sobre el modelo. Una vez dichos elementos forman parte del modelo, es posible recolocarlos para una mejor visualización a gusto del usuario manteniendo pulsado el botón izquierdo.

Una vez añadidos estos elementos vamos a modificar sus nombres. Esto lo podemos hacer en las *Properties / General* de cada elemento. Los renombramos como: *timeUpdate* y *TimeHandler* respectivamente (los puertos deben tener la primera letra en minúscula).

A continuación vamos a importar al espacio de trabajo el proyecto *CommSampleObject*. En este repositorio se incluyen objetos de comunicación básicos como es el *CommSampleTime*.

Es posible importar únicamente el objeto que vayamos a utilizar en nuestro diseño en lugar de hacerlo con el proyecto o repositorio completo. Sin embargo, debido a que han surgido algunos errores cuando se han importado objetos sueltos, y a la comodidad que supone importar el proyecto completo, decidimos realizarlo de este modo.

Lo hacemos de la siguiente manera:

- En la pestaña Outline: *Botón derecho sobre 'SamplePushTimedServer' / Smart Robotics / Import CommObject Repository.* A continuación buscamos el *'CommSampleObject'*.

Para comprobar que el proceso se ha realizado correctamente, debido a un problema de SmartSoft, será necesario cerrar el proyecto y volver abrirlo.

El siguiente paso será proceder a configurar el puerto. Para ello accedemos al perfil del mismo: *Properties / Profile*. Fijamos los siguientes valores:

- *smartPushTimedHandler : smartPushTimedHandler [1..1] = TimeHandler*
- *cycle: Integer [1..1] = 500*
- *timeUnit : TimeUnitKind [1..1] = ms*
- *commObject : Class [1..1] = CommSampleTime*
- *serviceName : String [1..1] = time*

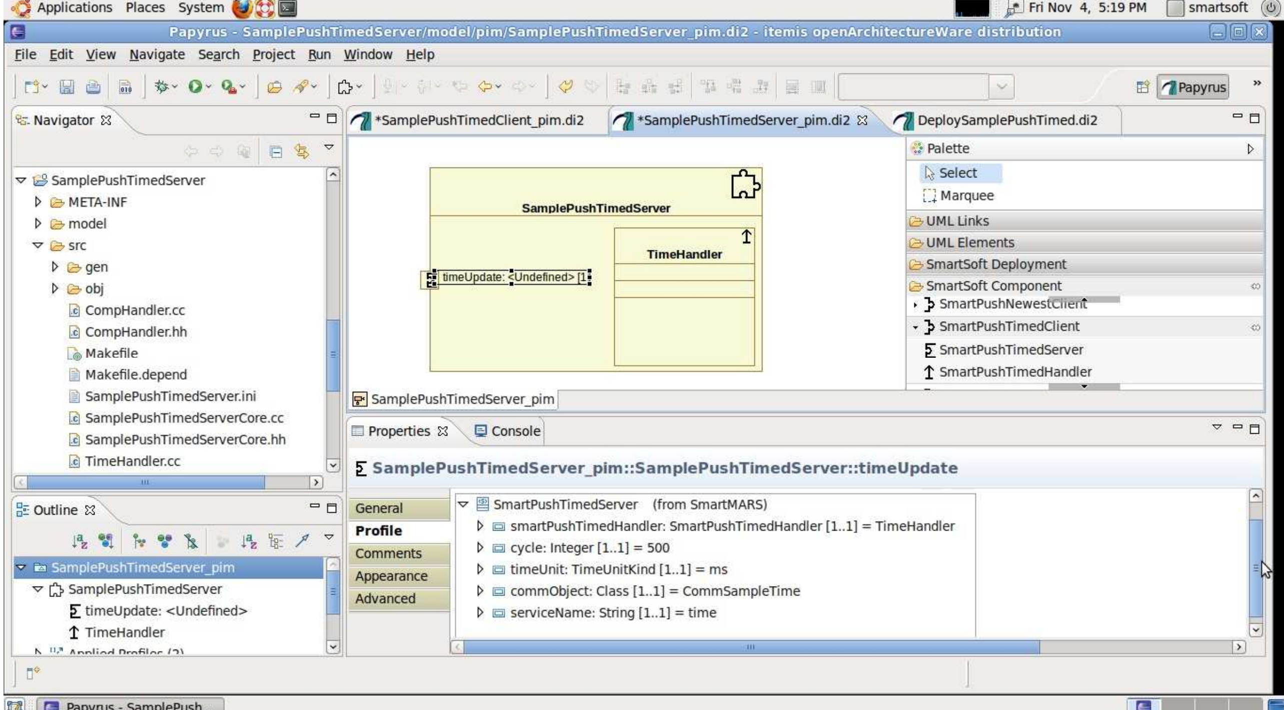


Figura 5.17: Modelado del componente SamplePushTimedServer

Vemos de esta manera, como se vincula el puerto de comunicación con el handler; así como la manera en la que se define el tipo de objeto de comunicación que utiliza el mismo. Con este tipo de patrón de comunicación podemos especificar cada cuanto queremos que el handler entre en juego (cantidad y unidad de tiempo). Por último establecemos el identificador que tendrá el servicio establecido entre los componentes.

El siguiente paso es generar el código asociado al modelo. Para ello, hacemos lo siguiente:

- *Botón derecho sobre el proyecto / Smart Robotics / Run Code Generator*

Una vez hemos realizado esto, en la carpeta src de nuestro proyecto aparecen distintos ficheros. En este caso, nosotros estamos interesados en el fichero *TimeHandler.cc*. En este fichero se detalla el comportamiento de la tarea.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
void TimeHandler::handlePushTimer(
        CHS::PushTimedServer<CommSampleObjects::CommSampleTime> & server) throw(){

        time_t time_now;
        struct tm *time_p;
        CHS::StatusCode status;
        int i=0;

        CommSampleObjects::CommSampleTime a;

        time_now = time(0);
        time_p = gmtime(&time_now);

        a.set(time_p->tm_hour,time_p->tm_min,time_p->tm_sec);
        status = server.put(a);

       std::cout << "<push timed data> " << i++ << " status: " <<
       CHS::StatusCodeConversion(status) << " ";a.print();
}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Como se ha dicho, el handler se ejecutará cada 500ms. Su ejecución se a preparar el objeto *CommSampleTime* y enviarlo a los clientes suscritos mediante la función *put.* Para seguir la ejecución de una manera más clara, el servidor imprime por pantalla el objeto enviado.

Ya solo nos queda realizar el 'build' sobre el proyecto:

- *Botón derecho sobre el proyecto / Smart Robotics / Build Project*

Construcción del componente cliente “SamplePushTimedClient”

Creamos un proyecto nuevo:

- *File / New / Project → Smart Robotic Project / SmartSoft Component →* Fijamos el nombre del componente. (p.ej '*SamplePushTimedClient*')

Abrimos el modelo:

- *SampleSendClient / model / pim / SamplePushTimedClient_pim.di2*

Este componente está compuesto por un puerto *SmartPushTimedClient* y una tarea *SmartTask.* Ambos elementos los encontramos en el apartado *SmartSoft Component* dentro de *Palette.* Una vez añadidos estos elementos vamos a modificamos sus nombres en las *Properties* de cada elemento (*timeClient* y *TimeTask* respectivamente).

Al igual que hacíamos en el caso del servidor, importamos del espacio de trabajo el proyecto *CommSampleObject,* ya que utilizaremos uno de los objetos de comunicación contenidos en este proyecto. En concreto será el *CommSampleTime*, que también establecimos en el servidor.

El proceso es el siguiente:

- En la pestaña Outline: Botón derecho sobre *'SamplePushNewestServer' / Smart Robotics / Import CommObject Repository.* A continuación buscamos el *'CommSampleObject'*.

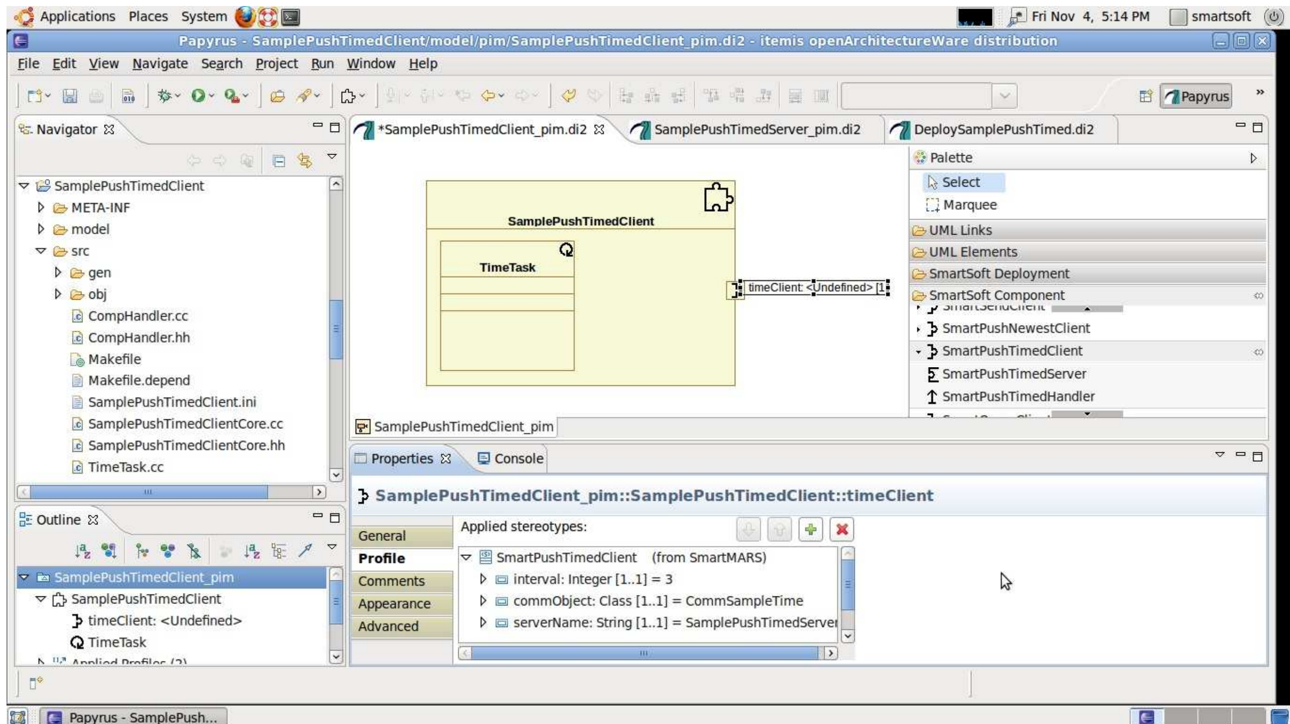


Figura 5.18: Modelado del componente SamplePushTimedClient

El siguiente paso será proceder a configurar el puerto. Para ello accedemos al perfil del mismo: *Properties / Profile*.

Configuración del puerto SamplePushTimedClient:

- *interval : Integer [1..1] = 3*
- *commObject : Class [1..1] = CommSampleTime*
- *serverName : String [1..1] = SamplePushTimedServer*
- *serviceName: String [1..1] = time*

Establecemos por tanto que el servidor de dicho servicio es el componente que hemos creado anteriormente, el *SamplePushTimedServer.* Del mismo modo, especificamos el servicio exacto del que queremos hacer uso, *time*. Al igual que hacíamos antes, el tipo de objeto de comunicación que utiliza este puerto es *CommSampleTime.*

El siguiente paso es generar el código asociado al modelo. Para ello, hacemos lo siguiente:

- *Botón derecho sobre el proyecto / Smart Robotics / Run Code Generator*

Una vez hemos realizado esto, en la carpeta *src* modificamos el fichero *TimeTask.cc.* En este fichero se establece el comportamiento de la tarea Time*Task*. El código es el siguiente:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
int TimeTask::svc(){
        CommSampleObjects::CommSampleTime a;
        CHS::StatusCode status;
        double cycle;
        bool    running;

        status = COMP->timeClient->getServerInfo(cycle,running);
        if (status != CHS::SMART_OK){
                std::cout << "getServerInfo  status " << CHS::StatusCodeConversion(status)
                << " not ok => abort" << std::endl;
```

```
            exit(1);
        }
        else{
            std::cout << "getServerInfo  status " << CHS::StatusCodeConversion(status) <<
                " cycleTime [seconds] " << cycle << " server started " << running <<
                std::endl;
        }

        while(1){
            status = COMP->timeClient->getUpdateWait(a);
            if (status != CHS::SMART_OK)  {
                std::cout << "blocking wait  status " << CHS::StatusCodeConversion(status)
                << " not ok => retry ..." << std::endl;
            }
            else{
                std::cout << "blocking wait  status " << CHS::StatusCodeConversion(status)
                << " ";a.print();
            }
        }
        return 0;
}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

La primera parte del código se encarga de informar del servidor al se va a suscribir, informando del ciclo con el que éste trabaja. Empieza entonces un bucle infinito en el que el cliente imprime los objetos que va recibiendo, quedando bloqueado entre recepciones. Recordar que el cliente atenderá los objetos con un intervalo igual a 3.

Por último realizamos el 'build' sobre el proyecto:

- *Botón derecho sobre el proyecto / Smart Robotics / Build Project*

Construcción del "DeploySamplePushTimed"

Creamos un nuevo proyecto del tipo deployment:

- *File / New / Project* → *Smart Robotic Project / SmartSoft Deployment* → Fijamos el nombre del componente. (p.ej '*DeploySamplePushTimed*'').

Abrimos el modelo. Para ello, buscamos en la pestaña *Navigator* el proyecto recién creado:

- *DeploySamplePushTimed / model / pim / DeploySamplePushTimed.di2*

Lo primero será importar al proyecto los componentes que van a formar parte de este.

- En la pestaña Outline: Botón derecho sobre *'DeploySamplePushTimed ' / Smart Robotics / Import Component*. A continuación buscamos el *SamplePushTimedClient* y *SamplePushTimedServer.*

Al igual que ya pasaba cuando importábamos el *CommSampleObject*, ahora también será necesario cerrar y volver abrir el proyecto para asegurarse de que todo se ha realizado correctamente.

El siguiente paso, es arrastrar desde la misma pestaña *Outline*, los componentes al modelo del deployment, así como los puertos que ambos poseen. Una vez hecho, pasamos a relacionar los componentes. Para ello hacemos uso de *'Connection'*, que se encuentra en el apartado *SmartSoft Deployment* dentro de la vista *Palette.*

A continuación lo que hacemos es introducir el *NamingService* en el deployment. Este elemento también lo encontramos en el apartado *SmartSoft Deployment* dentro de la vista *Palette*. Modificamos el profile:

- *ip : String [1..1] = 0.0.0.0*
- *port : Integer [1..1] = 12345*

Por último realizamos el Deploy del proyecto:

- *Botón derecho sobre el proyecto / Smart Robotics / Deploy*

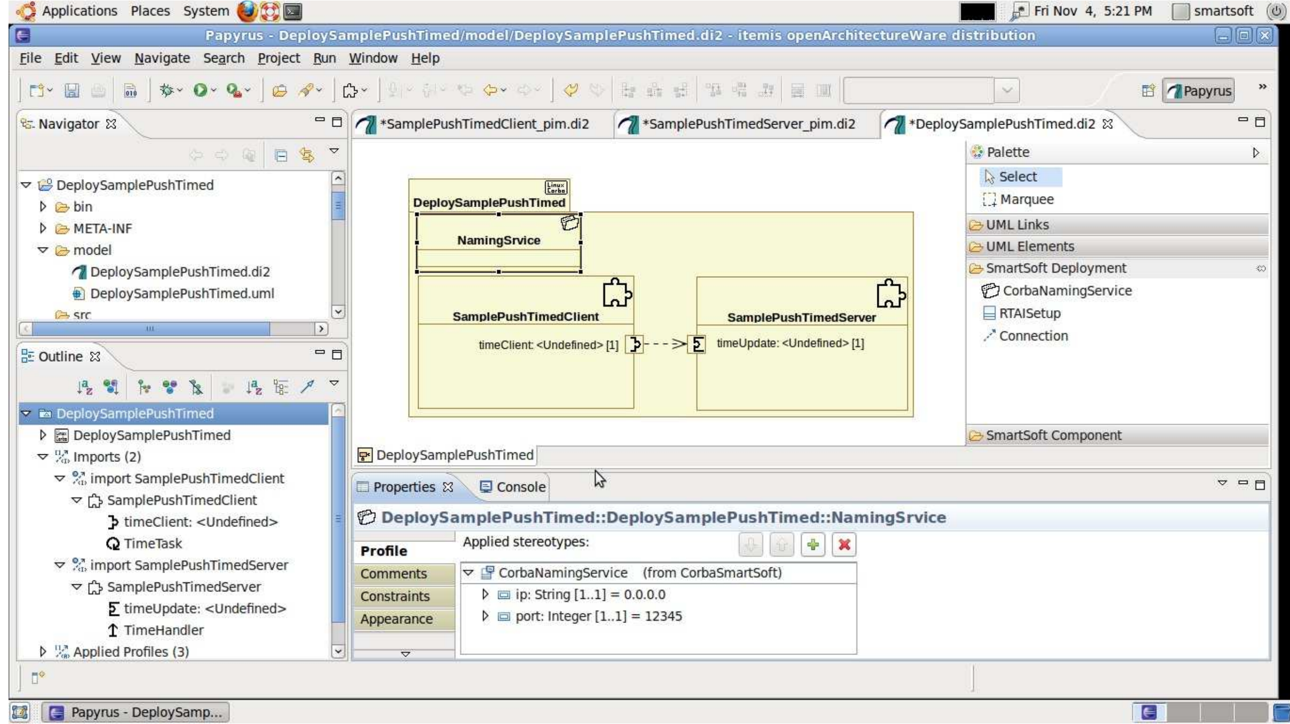


Figura 5.19: Modelado del deployment DeploySamplePushTimed

Tras esto, se ha creado en la carpeta *src* correspondiente al proyecto y denominada de igual manera que el deployment, el ejecutable *deploySamplePushTimed.sh.*

Para ejecutarlo escribimos en la consola:

```
>> ./deploySamplePushTimed start
```

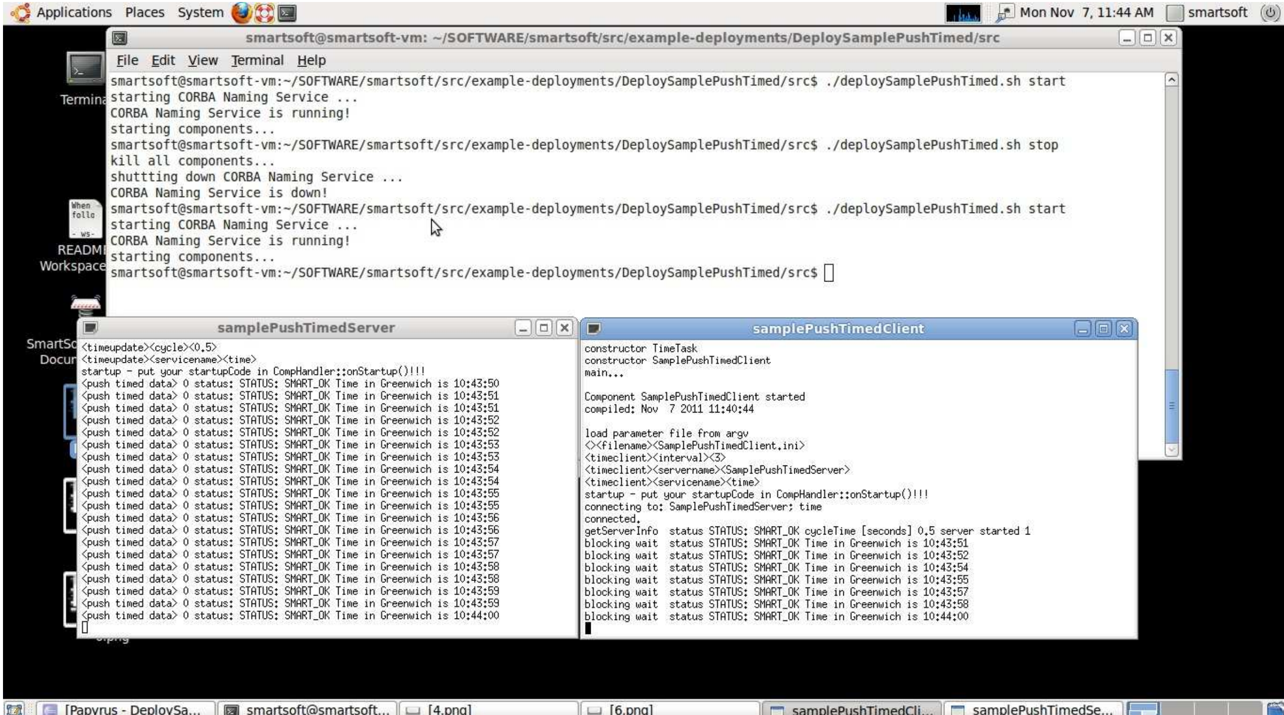


Figura 5.20: Ejecución de DeploySamplePushTimed

Los resultados son los esperados. El servidor realiza un envío cada 500 ms (se ajusta al segundo), y por tanto vemos 2 impresiones por segundo. Por otro, el cliente acepta las recepciones con un intervalo de 3 objetos, es decir; 1.5 segundos. Debido al redondeo que se realiza, aparece ese resultado por pantalla (1.5 + 1.5 = 1 + 2).

Para finalizar la ejecución del deployment introducimos el siguiente comando:

```
>> ./deploySamplePushTimed stop
```

6. DESARROLLO COMPLETO DE UNA APLICACIÓN

En este capítulo se describe el proceso de construcción de una aplicación robótica en la que el usuario introduce por teclado una “posición y ángulo objetivo” a la que el robot debe desplazarse de forma autónoma. La razón de diseñar esta aplicación en concreto es debido a que aún tratándose de una aplicación sencilla, se trata de una aplicación muy completa en la que el lector encontrará en ella todas las situaciones posibles a las que se puede enfrentar en el desarrollo de otra aplicación cualquiera.

En el apartado 6.1 diseñaremos la aplicación haciendo uso del simulador PlayerStage. En el apartado 6.2, tomando como punto de partida el diseño del apartado anterior, se describen los pasos necesarios para realizar la misma aplicación, pero utilizando esta vez el robot Pioneer P3-AT del que dispone el DSIE en el laboratorio en lugar del simulador.

6.1. Desarrollo de una aplicación robótica utilizando el simulador PlayerStage

En este capítulo se describe el proceso de diseño de una aplicación robótica completa haciendo uso de la herramienta *SmartSoftMDSD*. La finalidad de este trabajo es la construcción de una aplicación en la que haciendo uso del simulador *PlayerStage*, el usuario introduzca por teclado una “posición y ángulo objetivo” a la que el robot debe desplazarse de forma autónoma. La aplicación o deployment denominado *DeployGoTo,* está formada por los 3 componentes mostrados en la figura 6.1:

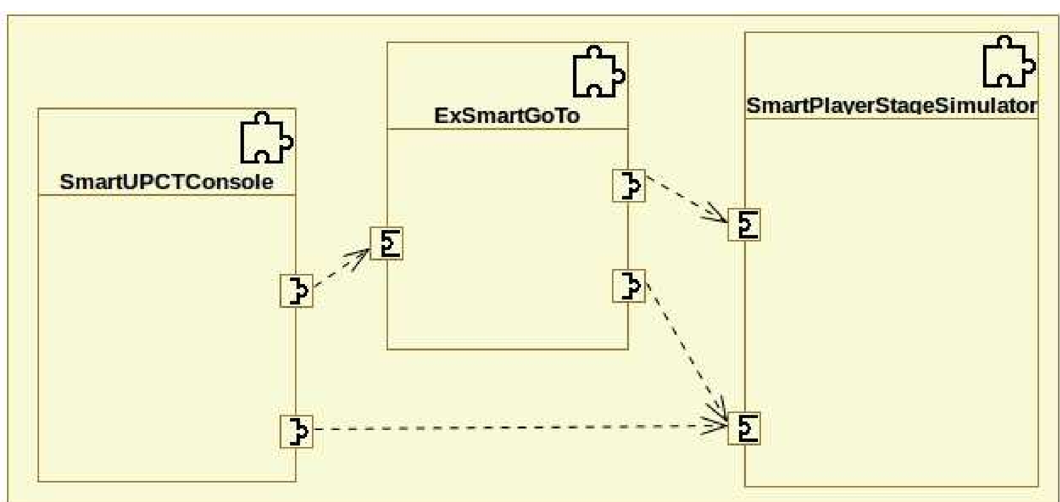


Figura 6.1: esquema general de la aplicación utilizando PlayerStage

- *Componente consola*
  Este componente denominado *SmartUPCTConsole* es una copia del componente *SmartRobotConsole*, a la que se le han realizado una serie de modificaciones añadiéndole funcionalidad, adaptándola a nuestra aplicación. Este componente hace uso de un nuevo objeto de comunicación *CommObjectGoTo* que contiene la posición y orientación objetivo. Además se encarga de imprimir por pantalla la posición y ángulo actual, así como la distancia al objetivo y la diferencia entre el ángulo actual y final.

- *Componente algoritmo*
  Este componente denominado *ExSmartGoTo* será diseñado desde cero y contiene toda la parte algorítmica de la aplicación. Su finalidad es que a través de la posición objetivo recibida en el objeto de comunicación *CommObjectGoTo* enviado por el *SmartUPCTConsole*, y de la posición y ángulo actual que recibe desde el componente simulador; calcular y enviar la velocidad lineal y angular con la que el robot debe desplazarse para alcanzar su objetivo.

- *Componente simulador*
  Este componente será el *SmartPlayerSimulator* que proporciona el equipo de desarrollo de SmartSoft. Nuestra labor en este caso consiste en analizar dicho componente y utilizar los puertos de comunicación que nos interesan del mismo. Los componentes *SmartUPCTConsole* y *ExSmartGoTo* deben entonces amoldarse a las especificaciones que les rige *SmartPlayerSimulator* en cuanto a los patrones y objetos de comunicación que utilizan los puertos utilizados.

Debido a que utilizaremos algunos proyectos que nos aporta el equipo de SmartSoft en el workspace *ws-mdsd,* parece lógico elegir este espacio de trabajo para llevar a cabo el diseño de la aplicación (si queremos hacerlo en otro workspace deberemos tener cuidado con los proyectos adicionales que será necesario importar). Esto quiere decir que todos los proyectos que se construyan se ubicarán en la carpeta *$SMART_ROOT / ws-mdsd* (ver apartado *4.4 – "Estructura de directorios").*

Por tanto las tareas a realizar en el proceso de desarrollo y puesta en marcha de la ejecución son:

- **Construcción del objeto de comunicación *CommObjectGoTo***
- **Copia del componente *SmartRobotConsole* → *SmartUPCTConsole***
- **Análisis del componente *SmartPlayerSimulator***
- **Modificación del *SmartUPCTConsole***
- **Construcción del componente *ExSmartGoTo***
- **Construcción del deployment *DeployGoTo***
- **Ejecución de la aplicación**

6.1.1 Construcción del objeto de comunicación *CommObjectGoTo*

En este apartado se describen los pasos necesarios para diseñar el objeto de comunicación *CommObjectGoTo*, que será enviado desde el componente consola *SmartUPCTConsole* hasta el componente *ExSmartGoTo*. Por tanto, este objeto de comunicación será el encargado de transmitir la información (parámetros) que el usuario de la aplicación introduce por teclado.

Siguiendo lo que se comentó en el apartado *"4.3.1 - Construcción de objetos de comunicación"*, para diseñar un objeto de comunicación, el primer paso es seleccionar el proyecto del tipo "repositorio de objetos de comunicación" en el que incluirlo. En este caso vamos a crear uno nuevo denominado *CommUPCTObjects:*

- *File / New Project / Smart Robotics Project – SmartSoft Communication Object Repository*

Una vez creado el repositorio, añadimos el objeto de comunicación a través del *CommObject* que encontramos en el apartado *SmartSoft CommObject* de la vista *Palette.* A continuación añadimos 3 tipos de datos primitivos mediante la opción *"Add new Property",* tal y como se muestra en la figura 6.2. Concretamente añadimos 3 *doubles* que se corresponden a la posición y orientación objetivo: coordenada X, coordenada Y, ángulo final. Para fijar el tipo de los atributos:

- *Seleccionar el atributo / vista Properties / General / Value Definition / Type.*

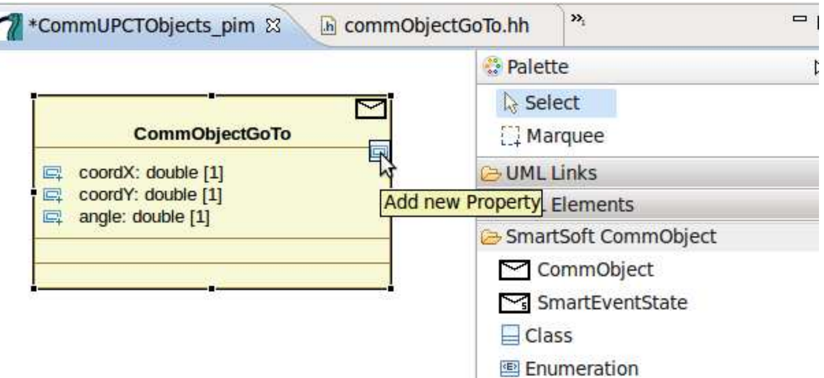


Figura 6.2: Modelado del objeto CommObjectGoTo

El siguiente paso es generar el código asociado al modelo mediante:

- *Botón derecho sobre el proyecto / Smart Robotics / Run Code Generator*

Una vez completado el proceso se han generado en la carpeta */src* del repositorio 2 ficheros denominados *CommObjectGoTo.hh* y *CommObjectGoTo.cc*. Estos son los ficheros donde incluiremos la interfaz de usuario del objeto de comunicación. Serán necesarias 3 funciones *set()* y 3 funciones *get()*, una por cada atributo. En este caso en lugar de hacer la declaración de las funciones en el fichero *CommObjectGoTo.hh* y su correspondiente implementación en *CommObjectGoTo.cc*, vamos a realizar la declaración e implementación en el *CommObjectGoTo.hh* mediante las funciones *inline*. Cuando hacemos esto, al compilar el código generado para la función se inserta en el punto donde se invoca a la función, en lugar de hacerlo en otro lugar y hacer una llamada. El uso de las funciones *inline* queda por tanto restringido a funciones de pequeño tamaño para evitar generar un ejecutable de tamaño considerable. En el fichero *CommObjectGoTo.hh* introducimos lo siguiente:

```
// user interfaces
	/**
	 Get the x coordinate of the position.
	 */
	inline double get_x() const {
		return idl_CommObjectGoTo.coordX;
	}
	/**
	 Get the y coordinate of the position.
	 */
	inline double get_y() const {
		return idl_CommObjectGoTo.coordY;
	}33993
	/**
	 Get the angle of the position.
	 */
	inline double get_angle() const {
		return idl_CommObjectGoTo.angle;
	}
	/**
	 Set the x coordinate of the position.
	 */
	inline void set_x(double x) {
		idl_CommObjectGoTo.coordX = x;
	}
	/**
	 Set the y coordinate of the position.
	 */
	inline void set_y(double y) {
		idl_CommObjectGoTo.coordY = y;
	}
	/**
	 Set the z coordinate of the position.
	 */
	inline void set_angle(double a) {
		idl_CommObjectGoTo.angle = a;
	}
```

Por último recompilamos el proyecto mediante:

- *Botón derecho sobre el proyecto / Smart Robotics / Build project*

Tras terminar el proceso satisfactoriamente, el objeto queda completado y el repositorio listo para ser importado por los componentes que deseen utilizar alguno de los objetos que contiene.

6.1.2. Copia del componente *SmartRobotConsole*

En el proceso de desarrollo de una aplicación robótica haciendo uso de la herramienta *SmartSoftMDSD*, se dará con bastante frecuencia la circunstancia en la que se desee copiar un proyecto ya existente y trabajar a partir de él. Puede ser para modificarlo y añadirle funcionalidad, para deshacernos de aquellas partes que no resultan interesantes, para basarnos en su modelo de cara a otro proyecto completamente nuevo…

En cualquier caso resulta interesante no estar obligado a modificar el proyecto de origen, pudiendo realizar una copia y trabajar sobre ella.

Eclipse permite realizar una copia de un proyecto del workspace asignándole como nombre por defecto "*Copyof*NameProject", que es posible modificar desde el apartado *Profile* de la vista *Properties*. Sin embargo estas copias mantienen el nombre del proyecto original en gran cantidad de ficheros, con lo que estos proyectos no son funcionales debido a incongruencias. Nos estamos refiriendo a proyecto en general y no a un tipo en concreto (repositorio de objetos, componente o deployment), ya que el proceso será idéntico en todos los casos.

Conseguir que estos proyectos sean funcionales pasa por reemplazar todas las coincidencias que encontremos con el nombre del proyecto original, por el nombre del nuevo proyecto. Por otro lado este proceso puede resultar algo tedioso e incómodo, con lo que será tarea del usuario decidir si le interesa copiar el proyecto o por el contrario repetirlo desde cero el mismo. El proceso de copia resulta por tanto conveniente en proyectos de gran envergadura que poseen gran cantidad de elementos (puertos, tareas, atributos, etc.), mientras que para proyectos sencillos puede resultar más conveniente realizar desde cero el modelo, general el código asociado, y completar los *hotspots* sirviéndonos del original.
A continuación se describen los pasos a seguir para realizar la copia de un proyecto denominado *SmartUPCTConsole*, tomando como componente de origen el *SmartRobotConsole* proporcionado por el equipo de desarrollo de SmartSoft. El proceso lo haremos en su gran mayoría desde una terminal, y consta de los siguientes pasos:

- ***Copiar el directorio del proyecto origen.***
- ***Eliminar aquellos ficheros que serán regenerados.***
- ***Renombrar todas las coincidencias.***
- ***Importar el proyecto al workspace.***

Copiar directorio del proyecto de origen

La copia debemos realizarla de la carpeta que contiene el proyecto completo. Estos proyectos se organizan como se vio anteriormente en el apartado "*4.4 – Estructura de directorios*". Las instrucciones que ejecutaremos en este caso serán:

```
smartsoft@smartsoft-vm:~$ cd $SMART_ROOT/src/components
smartsoft@smartsoft-vm:~/SOFTWARE/smartsoft/src/components$ cp -r SmartRobotConsole/ SmartUPCTConsole
smartsoft@smartsoft-vm:~/SOFTWARE/smartsoft/src/components$ cd SmartUPCTConsole/
smartsoft@smartsoft-vm:~/SOFTWARE/smartsoft/src/components/SmartUPCTConsole$ grep -r "SmartRobotConsole" * | wc -l
198
```

Comenzamos copiando la carpeta del proyecto *SmartRobotConsole*. Con el comando "`grep -r "SmartRobotConsole" *`" mostramos por pantalla las líneas de los ficheros donde existen coincidencias con la cadena *SmartRobotConsole* dentro del proyecto *SmartUPCTConsole*. El comando "`wc -l`", cuenta el número de líneas que se imprimen por pantalla. Es decir, ahora mismo existen 198 coincidencias de la cadena *SmartRobotConsole* en el proyecto *SmartUPCTConsole.* Debido al elevado número de coincidencias que aparecen resulta adecuado eliminar aquellas que puedan ser regeneradas de nuevo una vez que el componente sea funcional.

Eliminar aquellos ficheros que serán regenerados.

```
smartsoft@smartsoft-vm:~/SOFTWARE/smartsoft/src/components/SmartUPCTConsole$ rm -rf src/gen/ src/obj/
smartsoft@smartsoft-vm:~/SOFTWARE/smartsoft/src/components/SmartUPCTConsole$ grep -r "SmartRobotConsole" * | wc -l
133
smartsoft@smartsoft-vm:~/SOFTWARE/smartsoft/src/components/SmartUPCTConsole$ find -name .svn | xargs rm -rf
smartsoft@smartsoft-vm:~/SOFTWARE/smartsoft/src/components/SmartUPCTConsole$ grep -r "SmartRobotConsole" * | wc -l
53
```

Primero eliminamos los directorios *src/gen* y *src/obj*, ya que serán regenerados en el proceso de generación del código asociado del modelo desde la herramienta MDSD, una vez hayamos importado el componente al workspace. Por otro lado eliminamos todos los ficheros *.svn* que forman parte del sistema de control de

versiones. Tras esto, las coincidencias se reducen de 198 a 53. De cara a que sean más legibles las instrucciones debido a la longitud del *prompt*, movemos el proyecto al proyecto */home* del usuario.

```
smartsoft@smartsoft-vm:~/SOFTWARE/smartsoft/src/components/SmartUPCTConsole$ cd ..
smartsoft@smartsoft-vm:~/SOFTWARE/smartsoft/src/components$ mv SmartUPCTConsole/ /home/smartsoft/
smartsoft@smartsoft-vm:~/SOFTWARE/smartsoft/src/components$ cd
smartsoft@smartsoft-vm:~$ ls
Desktop  Downloads  SmartUPCTConsole  SOFTWARE  src  ws-examples  ws-mdsd  ws-toolchain  zafh.jpg
```

Renombrar las coincidencias

A la hora de renombrar las coincidencias que aparecen en los distintos ficheros se pueden dar 2 posibilidades: que el fichero también deba ser también renombrado o que no sea necesario. Parte de los ficheros que deben ser renombrados son los que pertenecen al modelo del componente (ficheros *.uml* y *.di2*), y que se encuentran en la carpeta *model/pim* (al modelo del componente siempre se le asigna el nombre del proyecto). Por otro lado, en la carpeta */src* de todo componente aparecen los ficheros *"Core.cc", "Core.hh"* y *".ini"* que también toman el nombre con el que se haya definido el proyecto. Para renombrar tanto las coincidencias como los nombres de los ficheros hacemos lo siguiente:

```
smartsoft@smartsoft-vm:~/SOFTWARE/smartsoft/src/components/SmartUPCTConsole$ cd ..
smartsoft@smartsoft-vm:~/SOFTWARE/smartsoft/src/components$ mv SmartUPCTConsole/ /home/smartsoft/
smartsoft@smartsoft-vm:~/SOFTWARE/smartsoft/src/components$ cd
smartsoft@smartsoft-vm:~$ cd SmartUPCTConsole/model/pim/
smartsoft@smartsoft-vm:~/SmartUPCTConsole/model/pim$ ls
SmartRobotConsole_pim.di2  SmartRobotConsole_pim.uml  SmartRobotConsole_pim.uml.out.uml
smartsoft@smartsoft-vm:~/SmartUPCTConsole/model/pim$ grep -r "SmartRobotConsole" * | wc -l
33
smartsoft@smartsoft-vm:~/SmartUPCTConsole/model/pim$ sed s/SmartRobotConsole/SmartUPCTConsole/g
SmartRobotConsole_pim.di2 > SmartUPCTConsole_pim.di2
smartsoft@smartsoft-vm:~/SmartUPCTConsole/model/pim$ sed s/SmartRobotConsole/SmartUPCTConsole/g
SmartRobotConsole_pim.uml > SmartUPCTConsole_pim.uml
smartsoft@smartsoft-vm:~/SmartUPCTConsole/model/pim$ sed s/SmartRobotConsole/SmartUPCTConsole/g
SmartRobotConsole_pim.uml.out.uml > SmartUPCTConsole_pim.uml.out.uml
smartsoft@smartsoft-vm:~/SmartUPCTConsole/model/pim$ rm SmartRobotConsole_pim.di2 SmartRobotConsole_pim.uml
SmartRobotConsole_pim.uml.out.uml
smartsoft@smartsoft-vm:~/SmartUPCTConsole/model/pim$ ls
SmartUPCTConsole_pim.di2  SmartUPCTConsole_pim.uml  SmartUPCTConsole_pim.uml.out.uml
smartsoft@smartsoft-vm:~/SmartUPCTConsole/model/pim$ grep -r "SmartRobotConsole" * | wc -l

0
smartsoft@smartsoft-vm:~/SmartUPCTConsole/model/pim$ cd ../../src
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ grep -r "SmartRobotConsole" * | wc -l
18
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ ls -l SmartRobotConsole
SmartRobotConsoleCore.cc  SmartRobotConsoleCore.hh  SmartRobotConsole.ini
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ sed s/SmartRobotConsole/SmartUPCTConsole/g SmartRobotConsoleCore.hh
> SmartUPCTConsoleCore.hh
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ sed s/SmartRobotConsole/SmartUPCTConsole/g SmartRobotConsoleCore.cc
> SmartUPCTConsoleCore.cc
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ sed s/SmartRobotConsole/SmartUPCTConsole/g SmartRobotConsole.ini >
SmartUPCTConsole.ini
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ rm -r SmartRobotConsoleCore.hh SmartRobotConsoleCore.cc
SmartRobotConsole.ini
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ ls SmartUPCTConsole
SmartUPCTConsoleCore.cc  SmartUPCTConsoleCore.hh  SmartUPCTConsole.ini
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ cd..
smartsoft@smartsoft-vm:~/SmartUPCTConsole$ grep -r "SmartRobotConsole" * | wc -l
14
```

Utilizamos el comando *sed* para intercambiar todas las cadenas "*SmartRobotConsole*" por "*SmartUPCTConsole*". Vemos que todavía encontramos 14 coincidencias que son:

```
smartsoft@smartsoft-vm:~/SmartUPCTConsole$ grep -r "SmartRobotConsole" *
META-INF/MANIFEST.MF:Bundle-Name: SmartRobotConsole
META-INF/MANIFEST.MF:Bundle-SymbolicName: SmartRobotConsole; singleton:=true
src/PlannerNoPathEventTask.cc:#include "gen/SmartRobotConsole.hh"
src/Makefile:INI_FILES    =       SmartRobotConsole.ini
src/GoalEventTask.cc:#include "gen/SmartRobotConsole.hh"
src/Makefile.depend:obj/CompHandler.o: CompHandler.cc CompHandler.hh gen/SmartRobotConsole.hh \
src/Makefile.depend: gen/../SmartRobotConsoleCore.hh \
src/Makefile.depend: gen/SmartRobotConsole.hh gen/../SmartRobotConsoleCore.hh \
src/Makefile.depend: gen/SmartRobotConsole.hh gen/../SmartRobotConsoleCore.hh \
src/Makefile.depend: gen/SmartRobotConsole.hh gen/../SmartRobotConsoleCore.hh \
src/Makefile.depend:obj/SmartRobotConsoleCore.o: SmartRobotConsoleCore.cc \
src/Makefile.depend: SmartRobotConsoleCore.hh /usr/include/c++/4.4/iostream \
src/CompHandler.cc:#include "gen/SmartRobotConsole.hh"
src/ConsoleTask.cc:#include "gen/SmartRobotConsole.hh"
```

Estos ficheros no tendrán que ser renombrados como pasaba anteriormente. Algunos de ellos son comunes a todos los proyectos (*MANIFEST, Makefile, CompHandler.cc…*), mientras que otros son específicos de este (*ConsoleTask y GoalEventTask*). Lo haremos de la siguiente forma:

```
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ cp -r CompHandler.cc CompHandler.cc2
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ sed s/SmartRobotConsole/SmartUPCTConsole/g CompHandler.cc2 >
CompHandler.cc
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ cp -r Makefile Makefile2
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ sed s/SmartRobotConsole/SmartUPCTConsole/g Makefile2 > Makefile
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ cp -r Makefile.depend Makefile.depend2
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ sed s/SmartRobotConsole/SmartUPCTConsole/g Makefile.depend2 >
Makefile.depend
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ cd ..
smartsoft@smartsoft-vm:~/SmartUPCTConsole$ cp -r META-INF/MANIFEST.MF META-INF/MANIFEST.MF2
smartsoft@smartsoft-vm:~/SmartUPCTConsole$ sed s/SmartRobotConsole/SmartUPCTConsole/g META-INF/MANIFEST.MF2 >
META-INF/MANIFEST.MF
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ rm -rf CompHandler.cc2 Makefile2 Makefile.depend2 META-
INF/MANIFEST.MF2
```

Todo lo ejecutado hasta este punto será necesario repetirlo en el todo proceso de copia de un componente, al ser ficheros comunes a todos ellos (claro está cambiando las cadenas a reemplazar). Modificando ahora los ficheros propios de este componente:

```
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ cp -r ConsoleTask.cc ConsoleTask.cc2
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ sed s/SmartRobotConsole/SmartUPCTConsole/g ConsoleTask.cc2 >
ConsoleTask.cc
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ cp -r GoalEventTask.cc GoalEventTask.cc2
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ sed s/SmartRobotConsole/SmartUPCTConsole/g GoalEventTask.cc2 >
GoalEventTask.cc
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ cp -r PlannerNoPathEventTask.cc PlannerNoPathEventTask.cc2
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ sed s/SmartRobotConsole/SmartUPCTConsole/g
PlannerNoPathEventTask.cc2 > PlannerNoPathEventTask.cc
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ rm -rf ConsoleTask.cc2 GoalEventTask.cc2 PlannerNoPathEventTask.cc2
smartsoft@smartsoft-vm:~/SmartUPCTConsole$ cd ..
smartsoft@smartsoft-vm:~/SmartUPCTConsole$ grep -r "SmartRobotConsole" * | wc -l
0
smartsoft@smartsoft-vm:~/SmartUPCTConsole$ grep -r "SmartUPCTConsole" * | wc -l
53
```

Por tanto según el comando *grep* hemos conseguido reemplazar todas las coincidencias de la cadena "*SmartRobotConsole*" por "*SmartUPCTConsole*". Sin embargo esto no es del todo cierto, ya que existen 2 ficheros ocultos comunes a todos los proyectos que mantienen el nombre original, en los que el comando grep no busca coincidencias. Estos ficheros son: *".project"* y *".cproject"*

```
smartsoft@smartsoft-vm:~/SmartUPCTConsole$ cp -r .project .project2
smartsoft@smartsoft-vm:~/SmartUPCTConsole$ sed s/SmartRobotConsole/SmartUPCTConsole/g .project2 > .project
smartsoft@smartsoft-vm:~/SmartUPCTConsole$ cp -r .cproject .cproject2
smartsoft@smartsoft-vm:~/SmartUPCTConsole$ sed s/SmartRobotConsole/SmartUPCTConsole/g .cproject2 > .cproject
smartsoft@smartsoft-vm:~/SmartUPCTConsole$ rm -rf .project2 .cproject2
```

Ya solo queda un detalle por realizar. SmartSoft siempre genera los ejecutables de los componentes con el nombre del proyecto ubicándolo en la carpeta *$SMART_ROOT/bin*. Sin embargo para diferenciarlo del directorio raíz del propio proyecto, en el *Makefile* del componente se le asigna al ejecutable el mismo nombre pero empezado en minúscula. Es necesario modificar esto ya que de lo contrario tendremos 2 componentes funcionales diferentes, que acaban generando el ejecutable con el mismo identificador *smartRobotConsole* sobrescribiéndose el uno al otro.

```
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ grep -r "smartRobotConsole" *
Makefile:CPNT_1_BIN =     smartRobotConsole
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ cp -r Makefile Makefile2
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ sed s/smartRobotConsole/smartUPCTConsole/g Makefile2 > Makefile
smartsoft@smartsoft-vm:~/SmartUPCTConsole/src$ rm -rf Makefile2
```

De esta manera hemos terminado el proceso de copia del componente y ya solo queda importarlo a un workspace de la herramienta *SmartSoftMDSD* para empezar a trabajar con él.

Importar el proyecto al workspace.

Una vez hemos renombrado todas las coincidencias el componente está listo para ser importado al espacio de trabajo desde donde podremos trabajar con él.

Para importar el componente al workspace:

- *File / Import / Existing Projects into Workspace / Next*

En la ventana que aparece debemos marcar la opción *"Select root directory"*, y a continuación buscar la ubicación del directorio en el que se encuentra el proyecto. Para asegurarnos que Eclipse ha sido capaz de reconocer el proyecto que queremos importar debe aparecer en la vista *"Projects"*. El proceso para el caso concreto de nuestro componente *SmartUPCTConsole* lo encontramos en la figura 6.3.

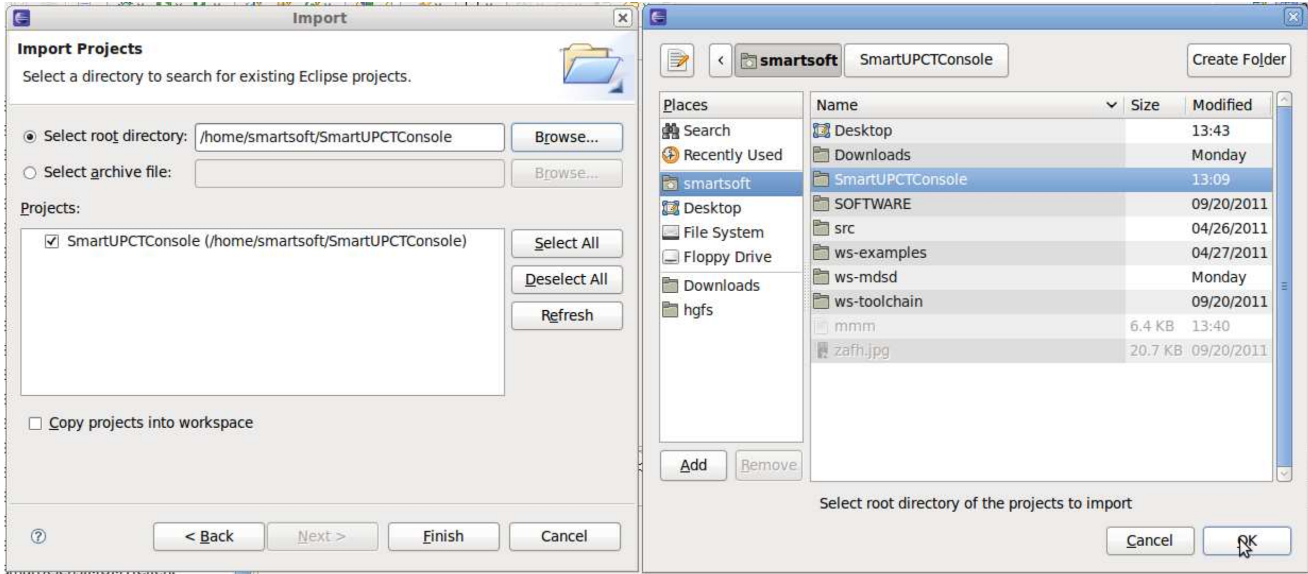


Figura 6.3: Importar proyecto al workspace

Por último comprobamos que el componente se ha importado correctamente al workspace y está listo para ser utilizado como vemos en la figura 6.4. Realizar la generación de código asociado al modelo es una posibilidad de comprobarlo.

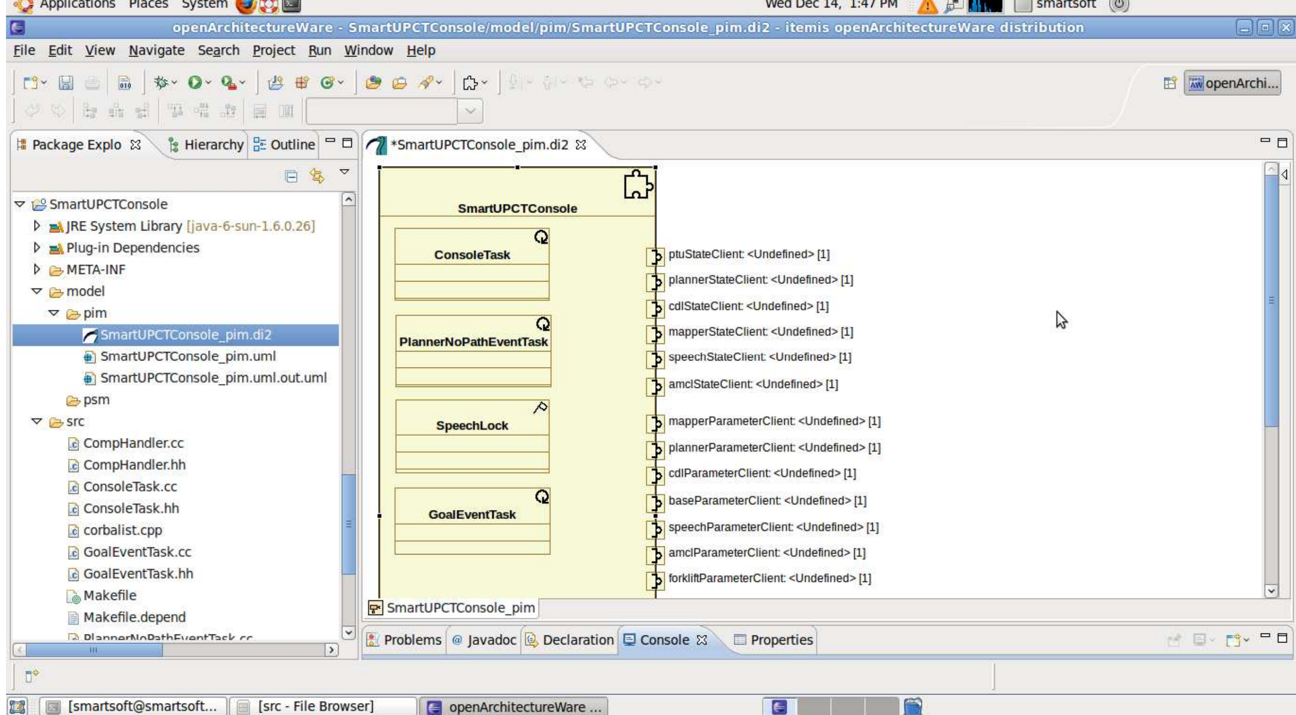


Figura 6.4: Componente SmartUPCTConsole

6.1.3. Análisis del componente *SmartPlayerStageSimulator*

El componente *SmartUPCTConsole* que hemos conseguido en el paso anterior, se encarga imprimir por pantalla la posición y ángulo actual del robot. Esta información la toma del componente simulador *SmartPlayerSimulator* a través de un puerto de comunicación, con lo que ambos componentes deben estar conectados. Además el componente *ExSmartGoTo* se sirve también de dicha información para realizar el cálculo de la velocidad lineal y angular con la que el robot debe desplazarse. Estas velocidades deben ser recibidas por el *SmartPlayerSimulator* a través de otro puerto de comunicación.

Debido a que no deseamos modificar el componente *SmartPlayerStageSimulator*, el desarrollo de los componentes *SmartUPCTConsole* y *ExSmartGoTo* se verán condicionados por el diseño que posee el componente simulador. Este es el motivo por el que resulta conveniente ver este apartado antes que la modificación del componente *SmartUPCTConsole*. El modelo del *SmartPlayerSimulator* aparece en la figura 6.5.

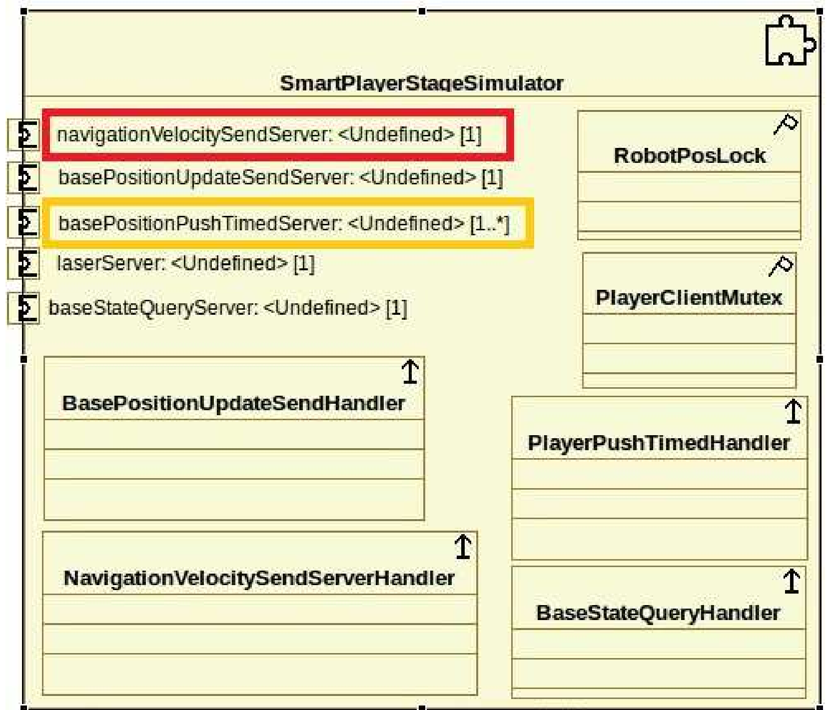


Figura 6.5: Modelo del componente SmartPlayerSimulator

Los puertos de comunicación que debemos utilizar son: *navigationVelocitySendServer* y *basePositionPushTimedServer* marcados en color rojo y amarillo respectivamente en la figura 6.5. *SmartUPCTConsole* hará uso del puerto *basePositionPushTimedServer*, mientras que *ExSmartGoTo* hará uso de ambos puertos. La información que debemos tener en cuenta para hacer uso de los servicios que ofrecen estos puertos es: el tipo de patrón que utilizan, los objetos de comunicación con los que trabajan y los nombres de los servicios que proporcionan.

- *navigationVelocitySendServer* → Puerto de comunicación de tipo *SmartSendServer* encargado de recibir la velocidad lineal y angular con la que el robot debe desplazarse. Utiliza el objeto de comunicación *CommNavigationVelocity*, y el servicio que utiliza se denomina *navigationvelocity*. El *Profile* de este puerto aparece en la figura 6.6.

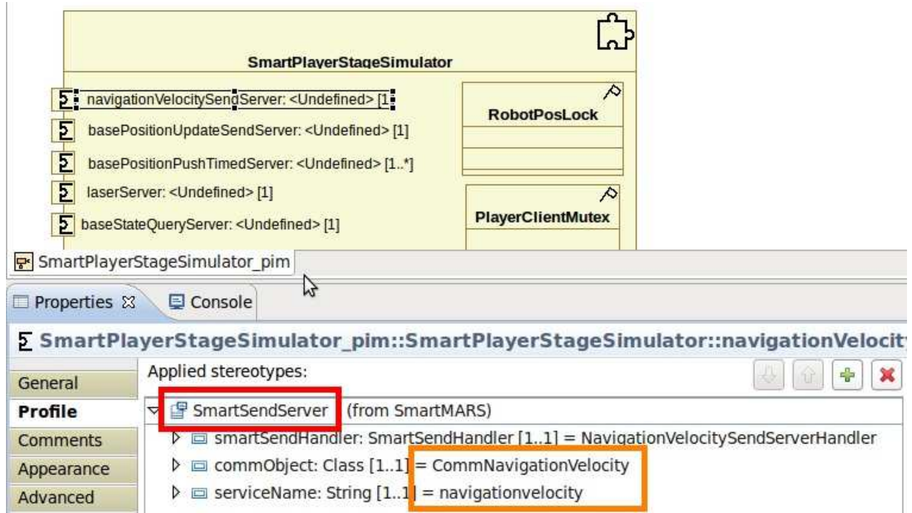


Figura 6.6: Puerto de comunicación navigationVelocitySendServer

- *basePositionPushTimedServer* → Puerto de comunicación de tipo *PushTimedServer* encargado de informar el estado actual del robot. Utiliza el objeto de comunicación *CommBaseState* y el servicio denominado *basestate*. El *Profile* de este puerto se muestra en la figura 6.7.

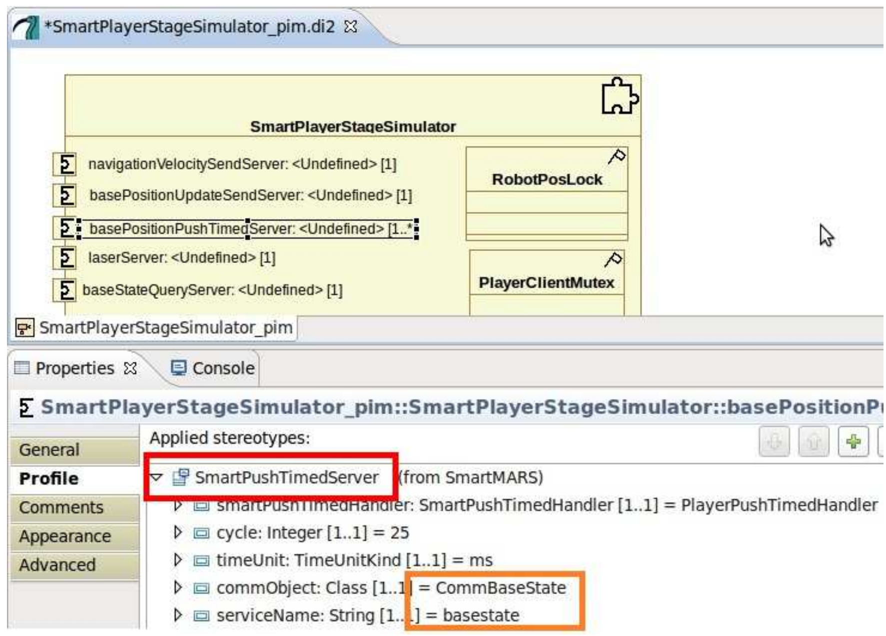


Figura 6.7: Puerto de comunicación basePositionPushTimedServer

6.1.4. Modificación del componente *SmartUPCTConsole*

En este apartado se describen los pasos necesarios para modificar el componente *SmartUPCTConsole* (copia del *SmartRobotConsole)* adaptándolo a nuestra aplicación. Para llevar a cabo este proceso, será necesario haber construido anteriormente el objeto de comunicación *CommObjectGoTo*, ya que este componente hará uso de dicho objeto. Ya que el componente toma la posición y ángulo actual del componente *SmartPlayerStageSimulator* a través del puerto *basePositionPushTimedServer*, las modificaciones que se llevarán a cabo se ven condicionadas por dicho puerto. El modelo original del componente se muestra en la figura 6.8.

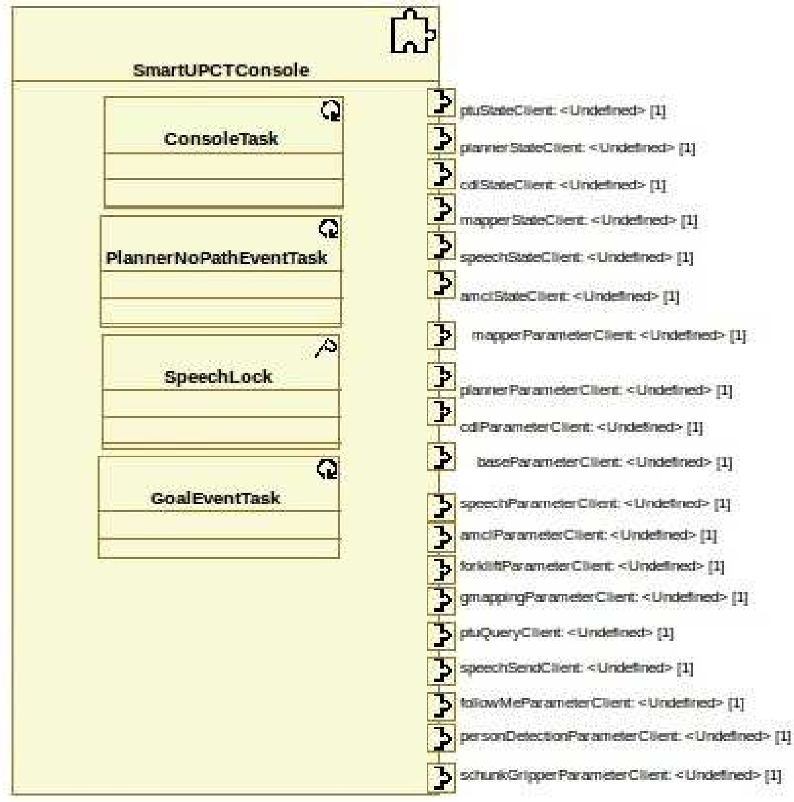


Figura 6.8: Modelo original del componente SmartUPCTConsole

Por la naturaleza de un componente consola este componente posee únicamente puertos de comunicación de tipo cliente. A continuación añadimos al modelo los 2 puertos clientes que necesitamos para nuestra aplicación.

- *positionToGoalClient* → Puerto de comunicación utilizado para enviar la posición y ángulo objetivo al componente *ExSmartGoTo*. Utiliza el patrón de tipo *ParameterClient,* el objeto de comunicación *CommObjectGoTo*, y el servicio utilizado se denomina *paramPosition.* El *Profile* de este puerto se muestra en la figura 6.9.

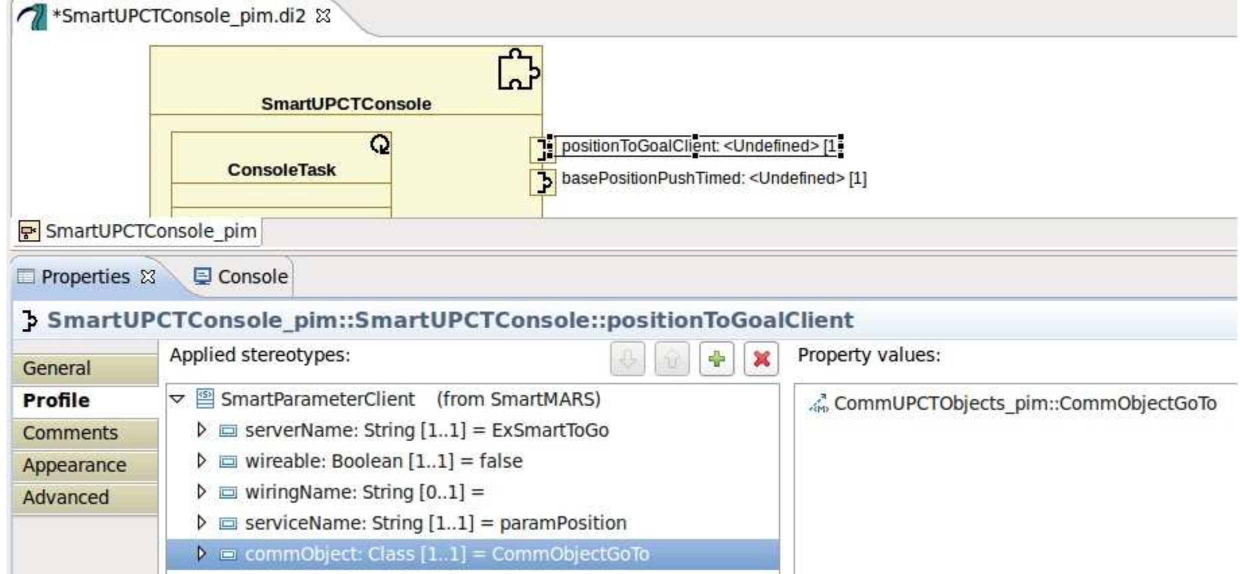


Figura 6.9: Puerto de comunicación positionToGoalClient

- *basePositionPushTimed* → Puerto de comunicación utilizado para recibir la posición y ángulo actual del robot en el *SmartPlayerStageSimulator,* del cual depende su configuración. Utiliza el patrón de comunicación de tipo *PushTimedClient*, el objeto de comunicación es el *CommBaseState* y el nombre del servicio utilizado es *basestate*. El *Profile* de este puerto se muestra en la figura 6.10.

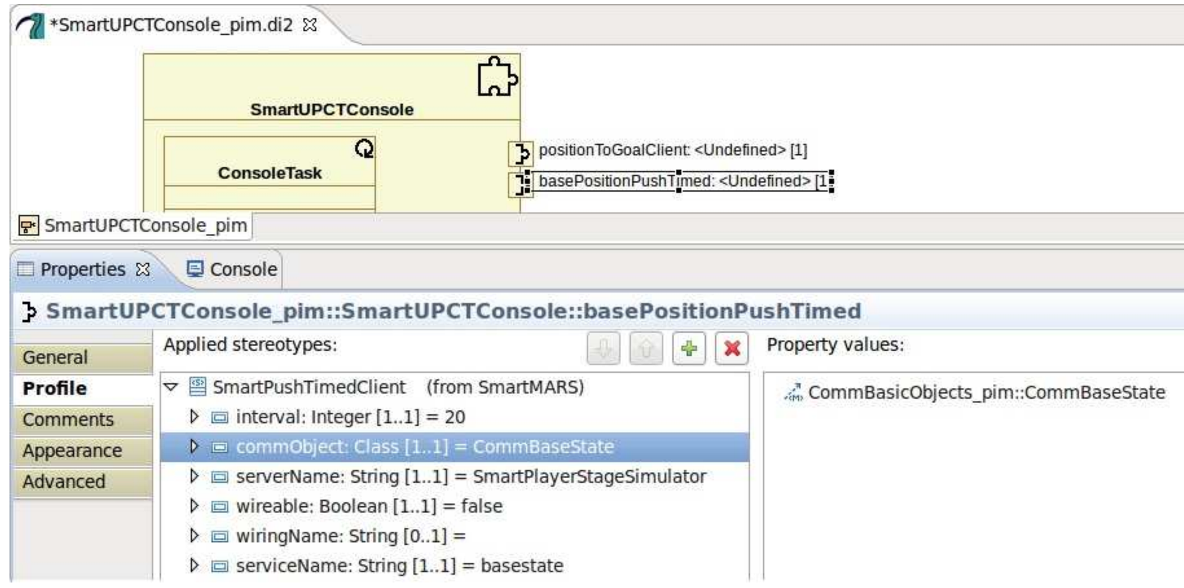


Figura 6.10: Puerto de comunicación basePositionPushTimed

El siguiente paso es generar el código asociado al modelo:

- *Botón derecho sobre el proyecto / Smart Robotics / Run Code Generator*

Una vez generado el código, el fichero a modificar es *consoleTask.cc*. Este fichero contiene un *"switch"* donde cada uno de los *"case"* se corresponde a una de las operaciones que la consola puede realizar (no es necesario por tanto definir una nueva tarea donde realizar el procesamiento). Por defecto *SmartUPCTConsole* muestra un menú con 22 opciones posibles. Por tanto establecemos un nuevo *"case 23"*. A continuación aparece el código introducido:

```
/////////////////////////////////////////////////////////////////////////////////////////
// 23 - ToGoal
  case 23:{

        CommUPCTObjects::CommObjectGoTo co;
        CommBasicObjects::CommBaseState dist;

        double Xpos, Ypos, Afinal, Xact, Yact, AngAct, dX, dY, distTiGoal=100, difA=1;

        cout << "Enter the X coordinate Y coordinate (in milimeters) and Angle (in degrees) of the
position you want to move the robot to (x y angle): ";

        cin >> Xpos >> Ypos >> Afinal;
        co.set_x(Xpos);
        co.set_y(Ypos);
        co.set_angle(Afinal);

        COMP->positionToGoalClient->send(co);

        while ((distToGoal>10) || (difA>0.1)){
               COMP->basePositionPushTimed->getUpdateWait(dist);
               Xact = dist.get_base_position().get_x();
               Yact = dist.get_base_position().get_y();
               AngAct = 360*dist.get_base_position().get_base_alpha() / (2*M_PI);
               if (AngAct < 0)
                      AngAct = 360 + AngAct;
               cout << "Posicion [X,Y] actual: [" << Xact << "," << Yact << "] ";
               dX = Xpos - Xact;
```

```
                dY = Ypos - Yact;
                distToGoal = sqrt((dX * dX) + (dY * dY));
                cout << " -  Distancia al objetivo: " << distToGoal << endl;
                if ((Afinal-AngAct) > 0)
                        difA = Afinal - AngAct;
                else
                        difA = AngAct - Afinal;
                cout << "Angulo actual: " << AngAct << "  -   Diferencia de angulos: " << difA <<
endl << endl;
        }
        break;
  }
/////////////////////////////////////////////////////////////////////////////////////////
```

Declaramos y rellenamos un objeto de comunicación del tipo *CommObjectGoTo* con los datos introducidos por teclado. Este objeto de comunicación será enviado al componente *ExSmartGoTo* a través del puerto *positionToGoalClient* utilizando la función *send* del patrón de comunicación.

Por otro lado se define un objeto de comunicación del tipo *CommBaseState* que será utilizado para recibir la posición y ángulo actual del simulador a través de la función *getUpdateWait* (función bloqueante) del puerto de comunicación *basePositionPushTimed*. Esto se realiza en un bucle *while*, cuya condición de parada es que la distancia al objetivo y la diferencia entre ángulos sea inferior a 2 tolerancias establecidas. Llegado ese punto acaba la ejecución del *"case"*, volviendo al menú principal pudiendo el usuario introducir un nuevo objetivo.

Por último compilamos el código comprobando que el código introducido es correcto:

- *Botón derecho sobre el proyecto / Smart Robotics / Build project*

6.1.5. Construcción del componente *ExSmartGoTo*

Este componente se encarga de recibir la posición y ángulo objetivo recibidos en un objeto de comunicación de tipo *CommObjectGoTo* enviado por el *SmartUPCTConsole* por un puerto que sigue el patrón *ParameterServer*. Además recibe la posición y ángulo actual del robot en el simulador en un objeto *CommBaseState* desde el componente *SmartPlayerStageSimulator* a través de un puerto que sigue el patrón *PushTimedClient*, para calcular la velocidad lineal y angular con la que el robot debe desplazarse para alcanzar su objetivo. La velocidad lineal y angular calculada, se envía al *SmartPlayerStageSimulator* a través de un puerto que sigue el patrón *SendClient* utilizando el objeto de comunicación *CommNavigationVelocity*. El modelo de este componente aparece en la figura 6.11.

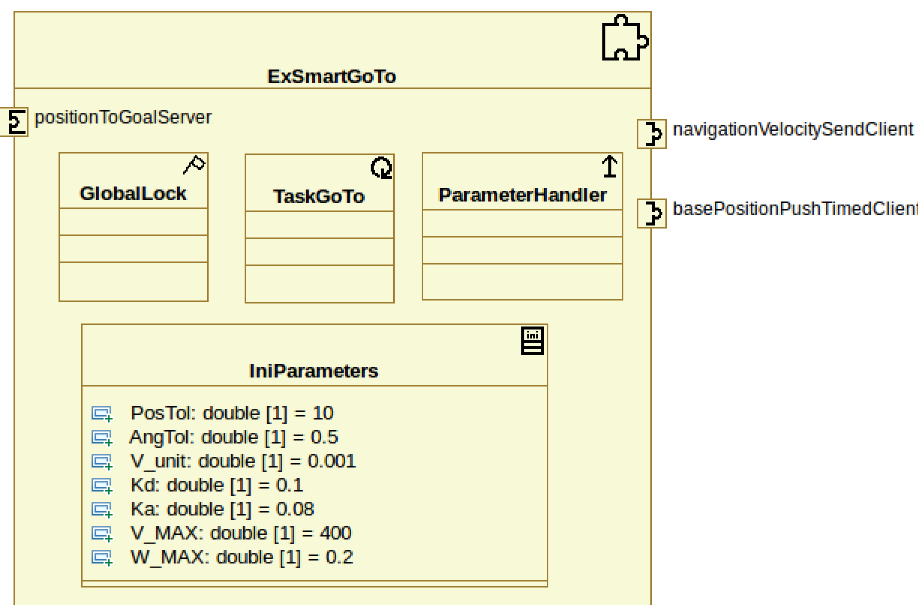


Figura 6.11: Modelo del componente ExSmartGoTo

Los elementos que encontramos en este modelo son:

- *GlobalLock* → *SmartMutex* utilizado para controlar el acceso en exclusión mutua a las variables globales del componente. Estas variables globales se corresponden a la posición y ángulo objetivo recibidas desde el *SmartUPCTConsole.*
- *TaskGoTo* → *SmartTask* donde se realiza la algoritmia característica del componente. Se utiliza la posición y ángulo objetivo y actual, para calcular la velocidad lineal y angular con la que el robot debe desplazarse.
- *ParameterHandler* → *SmartParameterHandler* utilizado para controlar el puerto servidor *positionToGoalServer*. Se encarga de actualizar el valor de las variables globales que más tarde lee la tarea *TaskGoTo*.
- *IniParameters* → *SmartIniParameterGroup* que contiene parámetros de inicialización del componente que son utilizados a modo de constantes: velocidades máximas, tolerancias y constantes necesarias para el cálculo de las velocidades. Estas variables deben ser utilizadas únicamente a modo de lectura (no escritura).
- *positionToGoalServer* → Puerto utilizado para recibir la posición y ángulo objetivos enviados desde el componente *SmartUPCTConsole*. La configuración de este puerto aparece en la figura 6.12.

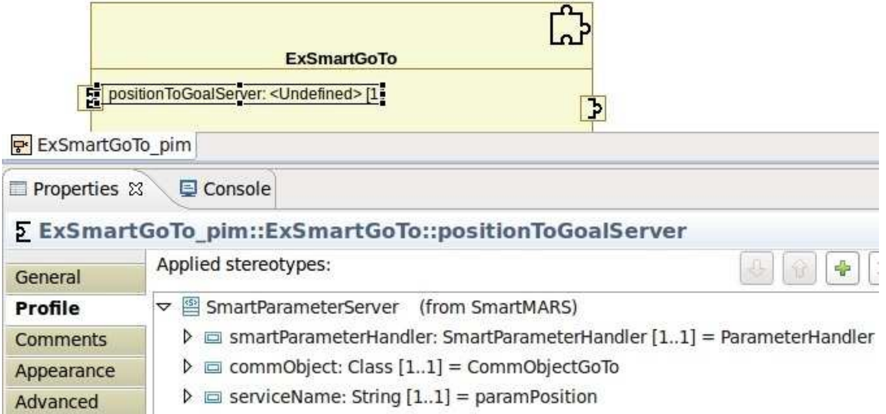


Figura 6.12: Puerto de comunicación positionToGoalServer

- *basePositionPushTimedClient* → Puerto de comunicación utilizado para recibir desde el componente *SmartPlayerStageSimulator* la posición y ángulo actual del robot en el simulador. La configuración de este puerto se muestra en la figura 6.13.

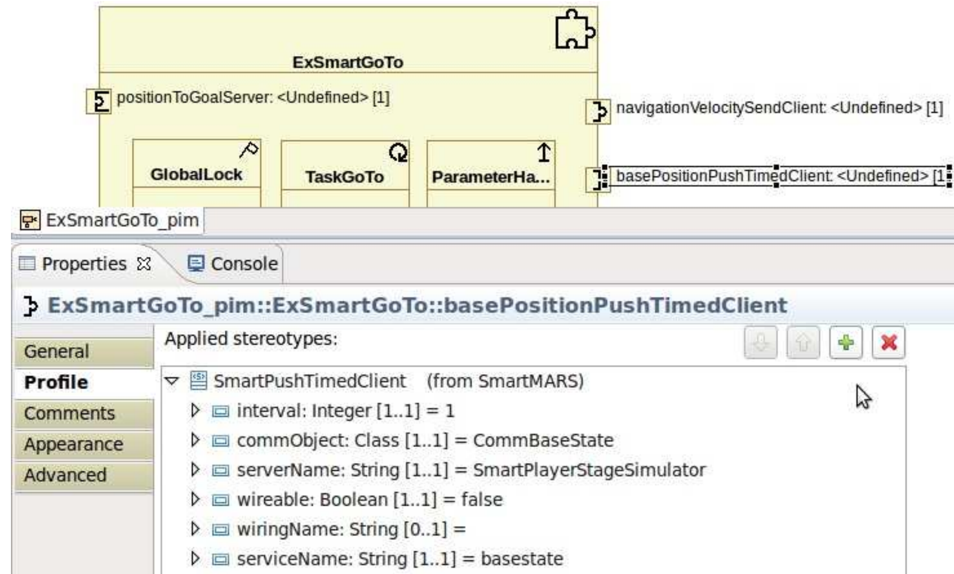


Figura 6.13: Puerto de comunicación basePositionPushTimedClient

- *navigationVelocitySendClient* → Puerto de comunicación utilizado para enviar al componente *SmartPlayerStageSimulator* la velocidad lineal y angular con la que el robot debe desplazarse para alcanzar su objetivo. La configuración de este puerto se muestra en la figura 6.14.

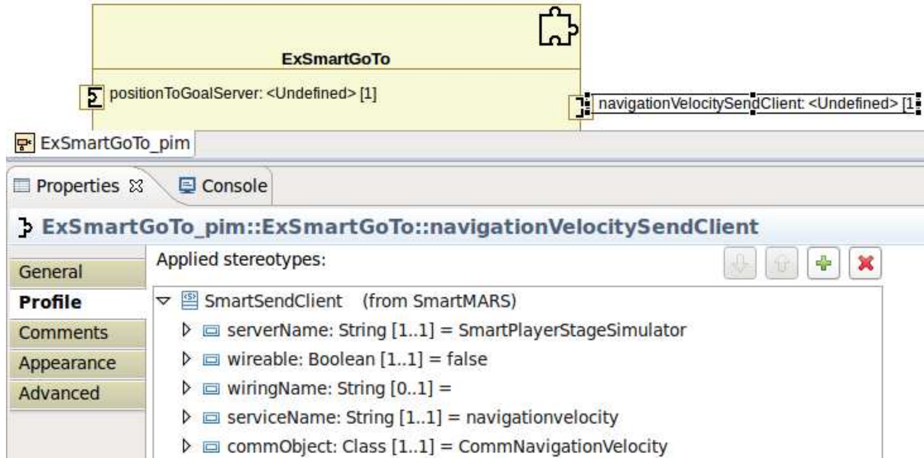


Figura 6.14: Puerto de comunicación navigationVelocitySendClient

Una vez hemos completado el modelo, generamos el código asociado:

- *Botón derecho sobre el proyecto / Smart Robotics / Run Code Generator*

La posición y ángulo objetivo se reciben a través del puerto *positionToGoalServer* y son capturados en su handler *ParameterHandler*. Sin embargo, estos valores son también utilizados en la tarea *TaskGoTo*. Aparece entonces la necesidad de utilizar variables globales al componente, de manera que sea posible acceder y modificar estas variables desde cualquier tarea o handler que posea. El fichero en el que debemos declarar las variables globales de forma que sea posible acceder a ellas desde cualquier punto del componente es en el *ExSmartGoToCore.hh*. Todo componente en el proceso de generación de código asociado al modelo, crea los ficheros "*Core.hh" y "*.Core.cc".

```
class ExSmartGoToCore{
private:

public:
        ExSmartGoToCore();
        double GoalPosX;
        double GoalPosY;
        double GoalAngle;
};
```

Definir estas variables dentro del elemento *IniParameter*s aunque posible, no es del todo correcto. Estas variables definidas en el *"*Core.hh"* serán modificadas a lo largo de la ejecución del programa en un número indeterminado de veces, mientras que la finalidad de las variables del SmartParameterGroup es ser únicamente de inicialización pudiendo modificar su valor desde el modelo de forma rápida, sin la necesidad de hurgar en los ficheros de código.

Una vez declaradas las variables en *ExSmartGoToCore.hh,* en el handler *ParameterHandler* se asignarán los parámetros recibidos desde el *SmartUPCTConsole* a dichas variables. Por otro lado, al tener una variable utilizada por distintas tareas, debemos procurar que el acceso a dicha variable se realice en exclusión mutua. Se hace entonces necesario un *SmartMutex*. El código necesario en el fichero *ParameterHandler.cc* es el siguiente:

```
void ParameterHandler::handleSend(const CommUPCTObjects::CommObjectGoTo &r) throw()
{
        COMP->GlobalLock.acquire();

        COMP->GoalPosX = r.get_x();
        cout << "Coordenada objetivo X: " << COMP->GoalPosX << std::endl;

        COMP->GoalPosY = r.get_y();
        cout << "Coordenada objetivo Y: " << COMP->GoalPosY << std::endl;
```

```
        COMP->GoalAngle = r.get_angle();
        cout << "Angulo objetivo: " << COMP->GoalAngle << std::endl;

        COMP->GlobalLock.release();
}
```

Llegado este punto podemos proceder a implementar el algoritmo en el fichero *TaskToGo*.

```
int TaskGoTo::svc() {

  CommBasicObjects::CommBaseState pos;
  CommBasicObjects::CommNavigationVelocity vel;
  double  XRobot, YRobot, AngleRobot, XFinal, YFinal, dX, dY, dAngle, v, w, Alpha, Distance;

  while (1) {

        //### Getting the inserted position
        COMP->GlobalLock.acquire();
        XFinal = COMP->GoalPosX;
        YFinal = COMP->GoalPosY;
        AngleFinal = COMP->GoalAngle;
        cout << "Posicion objetivo: " << XFinal << ", " << YFinal << ", " << AngleFinal <<
std::endl;
        COMP->GlobalLock.release();

        //### Getting actual robot position
        COMP->basePositionPushTimedClient->getUpdateWait(pos);
        XRobot = pos.get_base_position().get_x();    // Actual X position of the robot
        YRobot = pos.get_base_position().get_y();    // Actual Y position of the robot
        AngleRobot = (360*pos.get_base_position().get_base_alpha()) / (2*M_PI); // Actual angle
        if (AngleRobot < 0) { AngleRobot = 360 + AngleRobot; }

        //### Checking if there is any movement to be done
        dX = XFinal - XRobot;
        dY = YFinal - YRobot;
        Distance = sqrt((dX * dX) + (dY * dY));
        Alpha = AngleFinal - AngleRobot;
        if (Alpha > 180) { Alpha = Alpha - 360; }
        if (Alpha < -180) { Alpha = 360 + Alpha; }

        //### If yes execute the while code, if not look again for an inserted position
        if ((Distance>COMP->ini.IniParameters.PosTol)||(abs(Alpha)>COMP->ini.IniParameters.AngTol)){
           while (true) {
                COMP->basePositionPushTimedClient->getUpdateWait(pos);
                XRobot = pos.get_base_position().get_x();    // Actual X position of the robot
                YRobot = pos.get_base_position().get_y();    // Actual Y position of the robot
                AngleRobot = (360*pos.get_base_position().get_base_alpha()) / (2*M_PI);
                if (AngleRobot < 0) { AngleRobot = 360 + AngleRobot; }
                dX = XFinal - XRobot;
                dY = YFinal - YRobot;
                Distance = sqrt((dX * dX) + (dY * dY));
                Alpha = AngleFinal - AngleRobot;
                if (Alpha > 180) { Alpha = Alpha - 360; }
                if (Alpha < -180) { Alpha = 360 + Alpha; }
                if (Distance > COMP->ini.IniParameters.PosTol) {
                        if (dX != 0) {
                               dAngle = (360*atan(dY/dX)) / (2*M_PI);       // rad to deg
                               if (dAngle < 0) { dAngle = 360 + dAngle; }
                               if (dX < 0) {
                                      if (dY < 0) { dAngle = dAngle + 180; }
                                      if (dY > 0) { dAngle = dAngle - 180; }
                                      if (dY == 0) { dAngle= 180; }
                               }
                        } else {
                               if (dY > 0) { dAngle = 90; }
                               else { dAngle = 270; }
                        }
                        Alpha = dAngle - AngleRobot; // Rotates the robot
                        if (Alpha > 180) { Alpha = Alpha - 360; }
                        if (Alpha < -180) { Alpha = 360 + Alpha; }
                        v = COMP->ini.IniParameters.Kd * Distance;
                        if (v > COMP->ini.IniParameters.V_MAX) { v = COMP->ini.IniParameters.V_MAX; }
                        w = (COMP->ini.IniParameters.Ka * Alpha);
                        if (w > COMP->ini.IniParameters.W_MAX) { w = COMP->ini.IniParameters.W_MAX; }
                        vel.set_v(v, COMP->ini.IniParameters.V_unit);
                        vel.set_omega(w);
                        COMP->navigationVelocitySendClient->send(vel);
                }
```

```
            if ((Distance <= COMP->ini.IniParameters.PosTol) && (abs(Alpha) > COMP-
>ini.IniParameters.AngTol)) {
                    w = (COMP->ini.IniParameters.Ka * Alpha);
                    if (w > COMP->ini.IniParameters.W_MAX) { w = COMP->ini.IniParameters.W_MAX; }
                    vel.set_v(0, COMP->ini.IniParameters.V_unit);
                    vel.set_omega(w);
                    COMP->navigationVelocitySendClient->send(vel);
            }

            if ((Distance <= COMP->ini.IniParameters.PosTol) && (abs(Alpha) <= COMP-
>ini.IniParameters.AngTol)) {
                    vel.set_v(0, COMP->ini.IniParameters.V_unit);
                    vel.set_omega(0);
                    COMP->navigationVelocitySendClient->send(vel);
                    break;
            }

        }
    }
  }
  return 0;
}
```

En el código anterior encontramos los siguientes pasos:

- Tomar la posición y orientación objetivo que *ParameterHandler* se encargó de capturar y de fijar en las variables globales *XFinal, YFinal* y *AngleFinal.*
- Tomar la posición y ángulo actual del robot por el puerto *basePositionPushTimedClient* mediante la función bloqueante *getUpdateWait.*
- Comparación entre la distancia y ángulo objetivo con respecto a los actuales. Si se supera las tolerancias establecidas *PosTol* y *AngTol* será necesario desplazar el robot.
- Si se da la condición de que es necesario mover el robot entramos en un bucle con 3 condicionales. En todos los casos se envía una velocidad lineal y angular a través del puerto *navigationVelocitySendClient* utilizando la función *send*:
    - Si la distancia entre posiciones es mayor que la tolerancia *PosTol,* será necesario desplazar el robot, enviando las velocidades lineal v y angular w calculadas.
    - Si la distancia al objetivo es menor o igual a la tolerancia *PosTol*, pero la diferencia de ángulos es mayor a la tolerancia *AngTol.* Esto quiere decir que hemos alcanzado la posición en el espacio objetivo y resta orientar el robot, con lo que será necesario enviar al robot una velocidad lineal igual a 0 y una velocidad angular calculada.
    - Si las diferencias entre posiciones y ángulos son menores que las tolerancias enviamos una velocidad lineal y angular igual a 0 y terminamos el bucle.

Por último compilamos el código comprobando que el código introducido es correcto:
- *Botón derecho sobre el proyecto / Smart Robotics / Build project*

6.1.6. Construcción del deployment *DeployGoTo*

En este apartado se detalla el proceso de construcción del deployment donde relacionar los componentes *SmartUPCTConsole*, *ExSmartGoTo* y *SmartPlayerStageSimulator.* Comenzamos creando el proyecto:
- *File / New / Project → Smart Robotic Project / SmartSoft Deployment*

El siguiente paso será importar al proyecto los componentes que van a formar parte del deployment:
- Desde la vista Outline *Botón derecho sobre 'DeployGoTo' / Smart Robotics / Import Component. Seleccionamos los componentes SmartUPCTConsole*, *ExSmartGoTo* y *SmartPlayerStageSimulator*

A continuación desde la pestaña *Outline*, pinchamos sobre cada uno de los componentes y los arrastramos al deployment para hacerlos formar parte de él. Introducimos también el NamingService fijando una

configuración local (*Ip = 0.0.0.0, Port = 12345*). Por último comunicamos los componentes a través de los puertos de comunicación mediante el *'Connection'* de la vista *'Palette'*. El modelo finalmente queda como aparece en la figura 6.15.

Por último realizamos el proceso de deploy sobre el proyecto:

- *Botón derecho sobre el proyecto / Smart Robotics / Deploy*

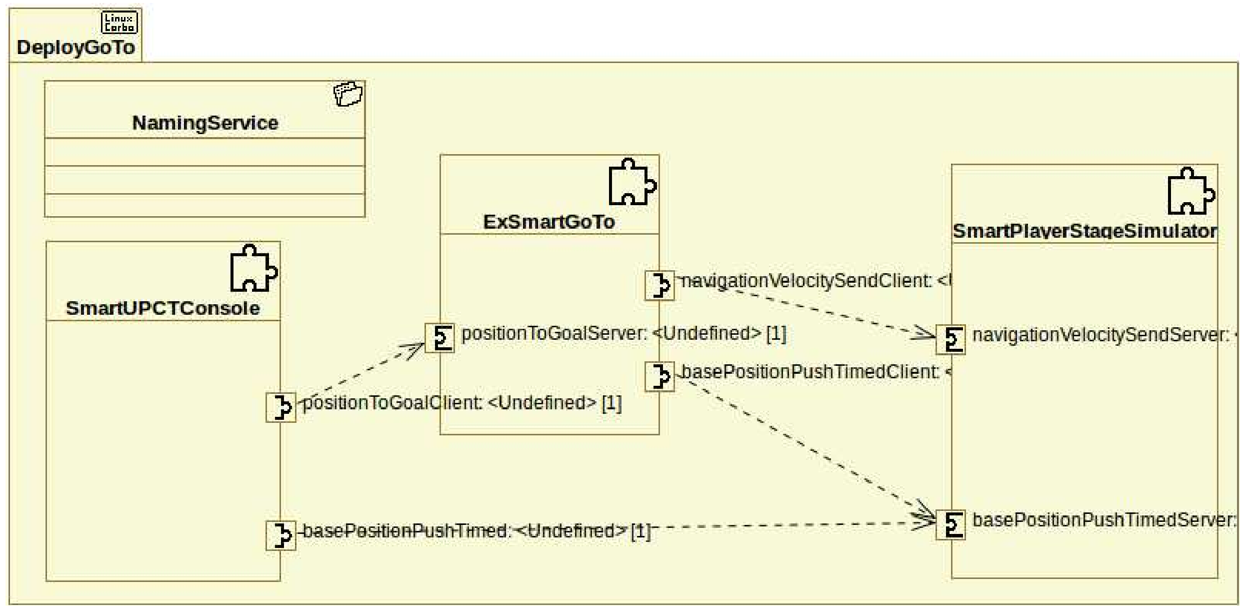


Figura 6.15: Modelo del deployment DeployGoTo

6.1.7. Ejecución de la aplicación

Como ya comentamos a comienzos del capítulo, hemos utilizado el workspace ws-mdsd con lo que todos los proyectos que hemos construido se encuentran en *$SMART_ROOT / src* (a excepción de *SmartUPCTConsole* que es una copia pero que nosotros trasladamos a dicho directorio). Esto en realidad no es algo necesariamente importante, ya que todos los ejecutables de los componentes se generan automáticamente en la carpeta *$SMART_ROOT / smartsoft / bin*, lugar donde por defecto todos los deployment los buscarán.

En el script generado asociado al deployment introducimos las siguientes instrucciones:

```
xterm -e "cd $SMART_ROOT/src/components/SmartPlayerStageSimulator/src/player_stage;
robot-player smart_laboratory.cfg" &
sleep 4
```

Con estas podemos fijar el mapa con el que deseamos que se ejecute nuestra aplicación. En este caso es *smart_laboratory* (los mapas se encuentran dentro de la carpeta */src* del componente *SmartPlayerStageSimulator*). Estas instrucciones se introducen antes de que el script ponga en marcha el NamingService de SmartSoft. Con todo esto ya estamos listos finalmente para ejecutar la aplicación:

```
smartsoft@smartsoft-vm:~/SOFTWARE/ws-mdsd/DeployGoTo/src$ ./deplyGoTo.sh start
```

Al ejecutar este comando aparecen una ventana por cada uno de los componentes que forman parte del deployment. El primer paso será introducir en la ventana correspondiente al componente *SmartUPCTConsole* la posición y orientación objetivo. Como aparece en la figura -, en este caso la posición y ángulo objetivo seleccionados son: coordenada *X = 3000*, coordenada *Y = 2000*, y *ángulo = 0º*. Nuestra aplicación toma como punto de partida el *[X,Y] = [0,0]* (así se fijo en el componente *ExSmartGoTo*), representándose las líneas del *grid* con una distancia entre ellas de 1000 mm. El robot debe avanzar por tanto 3 cuadrados hacia delante y 2 hacia arriba, para acabar orientado con el mismo ángulo que al inicio.

En la ventana del correspondiente al componente *ExSmartGoTo* se muestra la posición y orientación objetivo del robot en cada caso. Al comienzo el objetivo es la tupla *[0, 0, 0]*, con lo que el robot considera que ya se encuentra en el objetivo y no es necesario desplazarse. Por otro lado, en la ventana correspondiente al simulador se muestran la velocidad lineal y angular que recibe el componente *SmartPlayerStageSimulator* en cada instante. En la figura 6.16 se muestra la ejecución de la aplicación en el instante inicial.

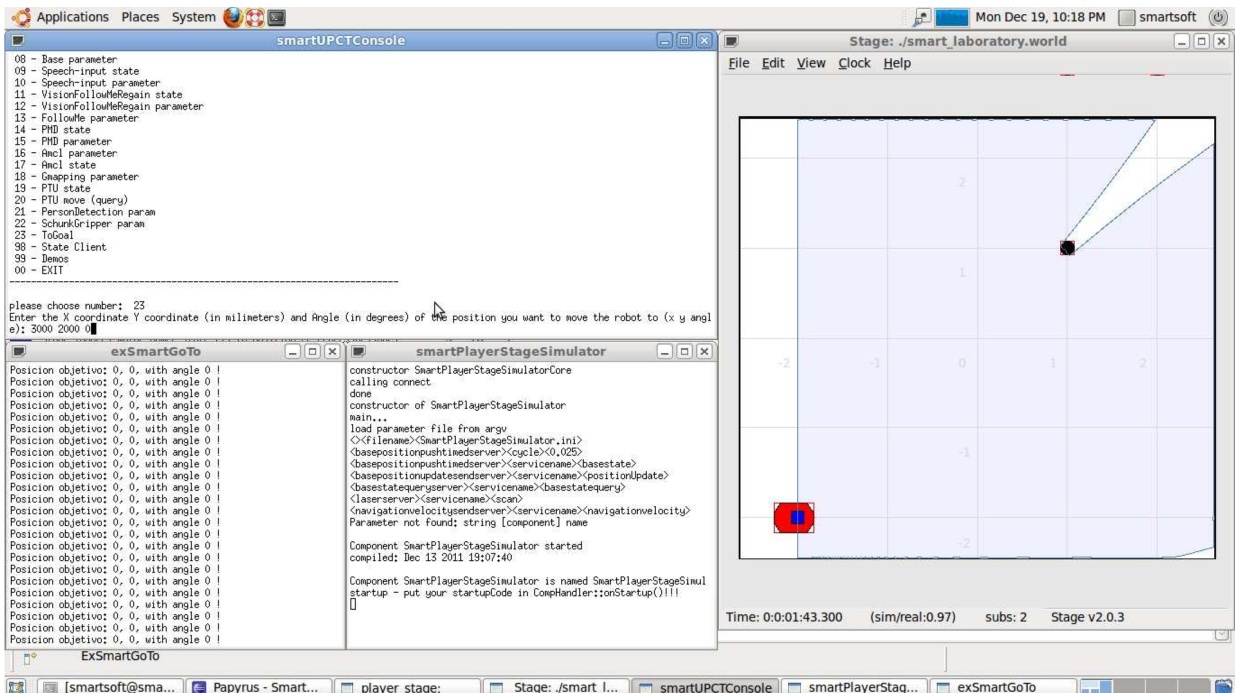


Figura 6.16: Ejecución de la aplicación utilizando el simulador PlayerStage – 1

Tras pulsar la tecla "Enter", el componente *ExSmartGoTo* captura el nuevo objetivo y lo muestra por pantalla. Este componente se encarga de calcular la velocidad lineal y angular con la que el robot debe desplazarse, que es enviada al *SmartPlayerStageSimulator* e imprimida por este. Este proceso se muestra en la figura 6.17.

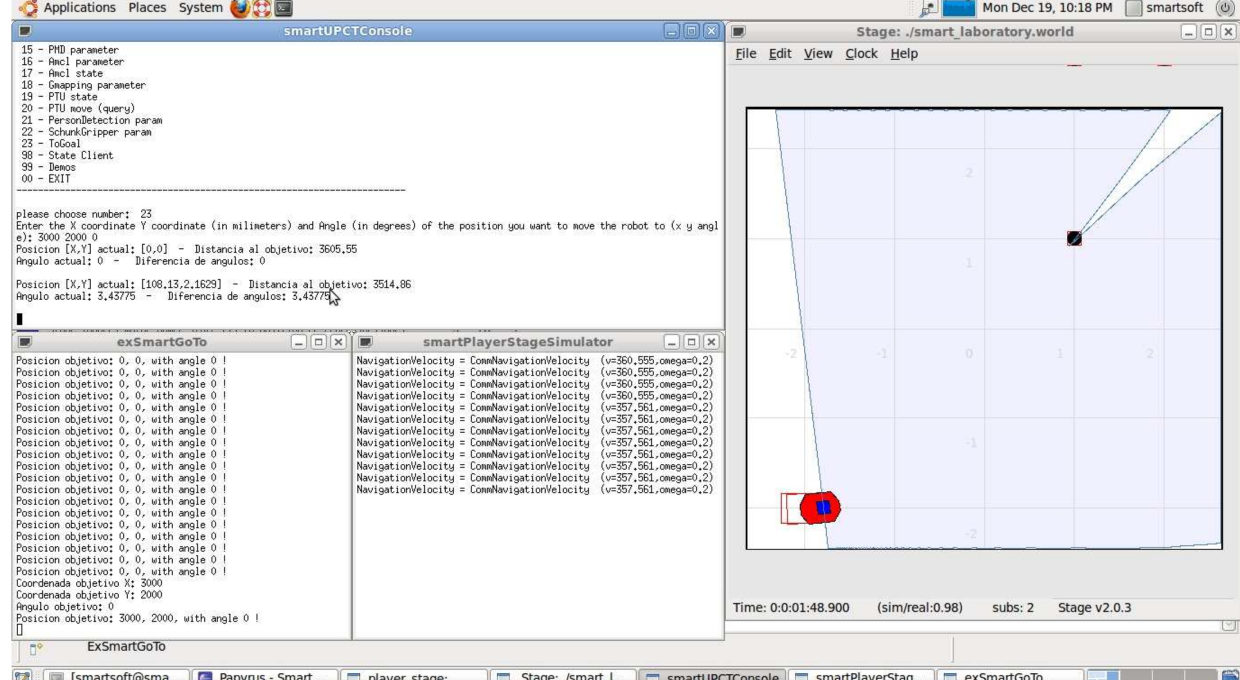


Figura 6.17: Ejecución de la aplicación utilizando el simulador PlayerStage - 2

Transcurridos aproximadamente unos 15 segundos (conforme se acerca va disminuyendo la velocidad), vemos como el robot alcanza su objetivo. Vemos como en la ventana del componente SmartUPCTConsole se retorna al menú principal, dando al usuario la opción de introducir un nuevo objetivo. Por otro lado *ExSmartGoTo* sigue informando de cuál es el objetivo, y en la ventana del *SmartPlayerStageSimulator* vemos como la velocidad lineal y angular que está recibiendo en este momento el robot es igual a 0.

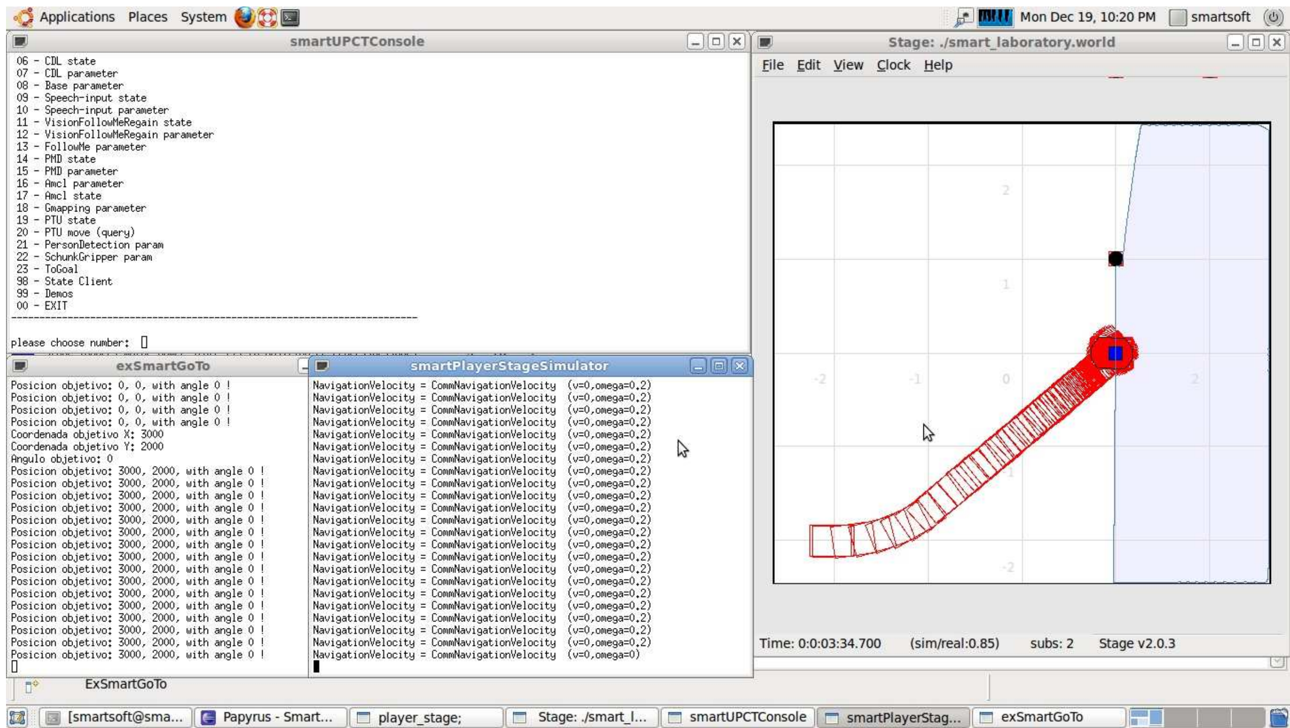


Figura 6.18: Ejecución de la aplicación utilizando el simulador PlayerStage - 3

Por último para detener la ejecución de la aplicación, desde la consola que la lanzamos introducimos lo siguiente:

```
smartsoft@smartsoft-vm:~/SOFTWARE/ws-mdsd/DeployGoTo/src$ ./deplyGoTo.sh stop
```

6.2. Desarrollo de una aplicación robótica utilizando el robot Pioneer P3-AT

En este apartado se describe el proceso de construcción de la aplicación anterior haciendo uso en este caso del robot real Pioneer P3-AT del laboratorio del DSIE en lugar del simulador PlayerStage. Recordemos que la aplicación consiste en que el robot debe alcanzar una posición y orientación objetivo que el usuario ha introducido por teclado.

De esta manera quedará claramente especificado el grado de reutilización de componentes que permiten alcanzar herramientas como SmartSoft. Tomando como punto de partida el deployment diseñado en el apartado anterior, únicamente será necesario intercambiar el componente simulador *SmartPlayerStageSimulator,* por otro componente que modele al robot real. Este componente es denominado *SmartPioneerBaseServer* y es proporcionado por el equipo de desarrollo de SmartSoft.

El deployment que caracteriza la aplicación estará formado por los componentes que aparecen en la figura 6.19

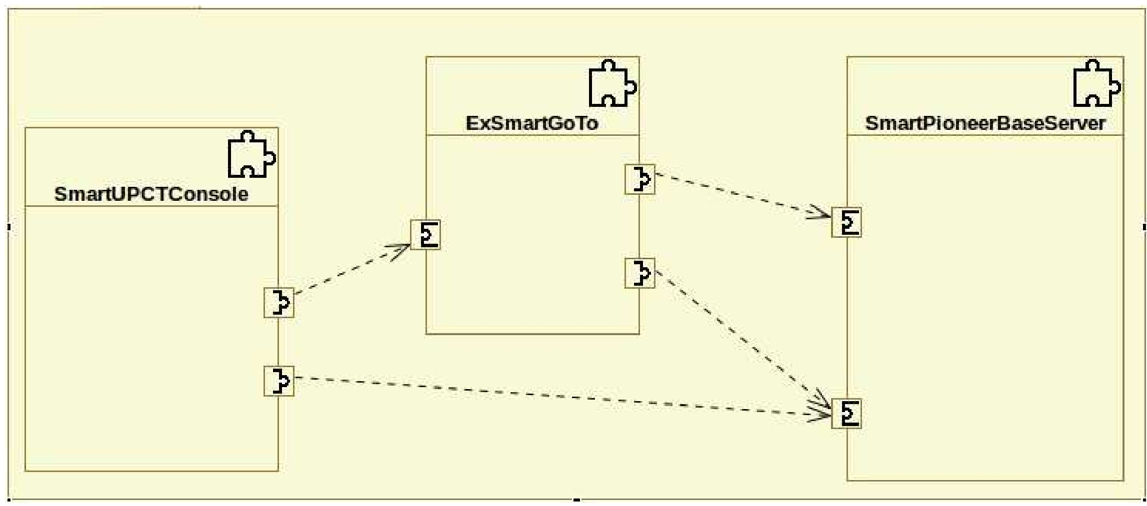


Figura 6.19: Esquema general de la aplicación utilizando el robot Pioneer

- *Componente consola*
  Este componente denominado *SmartUPCTConsole* es el mismo que se utilizó en el apartado anterior. Lo reutizaremos sin más.

- *Componente algoritmo*
  Este componente denominado *ExSmartGoTo* también es el mismo que se utilizó en el apartado anterior, con lo que lo igualmente lo reutilizaremos.

- *Componente robot*
  Este componente denominado *SmartPioneerBaseServer* es proporcionado por el equipo de desarrollo de SmartSoft, y es el encargado de modelar el robot real Pioneer. Como ya pasaba en el caso del *SmartPlayerStageSimulator* que también proporcionaba el equipo de SmartSoft, nuestra labor consiste en analizar dicho componente y utilizar los puertos de comunicación que nos interesan del mismo. Los componentes *SmartUPCTConsole* y *ExSmartGoTo* deben entonces amoldarse a las especificaciones que les rige *SmartPioneerBaseServer* en cuanto a los patrones y objetos de comunicación que utilizan los puertos utilizados.

Por tanto las tareas a realizar en el proceso de desarrollo y puesta en marcha de la ejecución, al reutilizar gran parte del diseño del apartado anterior y solo tener que intercambiar uno de los componentes, se reducen a las siguientes:

- **Análisis del componente *SmartPioneerBaseServer***
- **Modificación de los componentes en caso de ser necesario**
- **Construcción del deployment *DeployGoToRobot***
- **Ejecución de la aplicación**

6.2.1. Análisis del componente *SmartPioneerBaseServer*

En el apartado “6.1.3 - Análisis del componente *SmartPlayerStageSimulator”* realizábamos un estudio del componente simulador, ya que el diseño de los componentes *SmartUPCTConsole* y *ExSmartGoTo* se veían condicionados por el diseño del *SmartPlayerStageSimulator*.

En este caso, tampoco deseamos realizar cambio alguno en los componentes proporcionados por el equipo de desarrollo de SmartSoft. Partiendo entonces de los componentes *SmartUPCTConsole* y *ExSmartGoTo* ya implementados, realizamos un análisis del *SmartPioneerBaseServer* en busca de posibles diferencias de éste con respecto al componente simulador; para más tarde modificar en caso de ser necesario los componentes diseñados por nosotros. Encontramos documentación sobre este componente en la siguiente dirección http://smart-robotics.sourceforge.net/corbaSmartSoft/components.php?cmp=SmartPioneerBaseServer. El modelo del *SmartPioneerBaseServer* aparece en la figura 6.20.

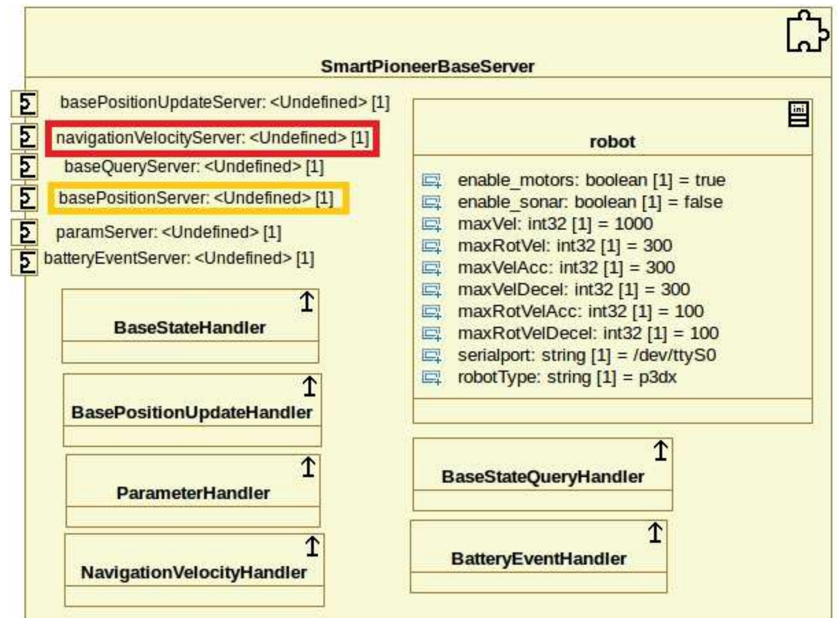


Figura 6.20: Modelo del componente SmartPioneerBaseServer

Los puertos de comunicación que debemos utilizar son: *navigationVelocityServer* y *basePositionServer* marcados en color rojo y amarillo respectivamente. Como observamos, los nombres de los puertos son prácticamente idénticos con respecto a los que tenían en el *SmartPlayerStageSimulator* (*navigationVelocitySendServer* pasa a ser *navigationVelocitySendServer*, y *basePositionPushTimedServer* pasa a ser *basePositionServer*). Para saber si debemos realizar algún cambio en los componentes tenemos que asegurarnos que los patrones que implementan estos puertos, los objetos de comunicación que utilizan, y el nombre con el que se identifica el servicio que prestan coinciden con los del apartado anterior.

- *navigationVelocityServer* → Puerto de comunicación de tipo *SmartSendServer* encargado de recibir la velocidad lineal y angular con la que el robot debe desplazarse. Utiliza el objeto de comunicación *CommNavigationVelocity*, y el servicio que utiliza se denomina *navigationvelocity*. El *Profile* de este puerto aparece en la figura 6.21.

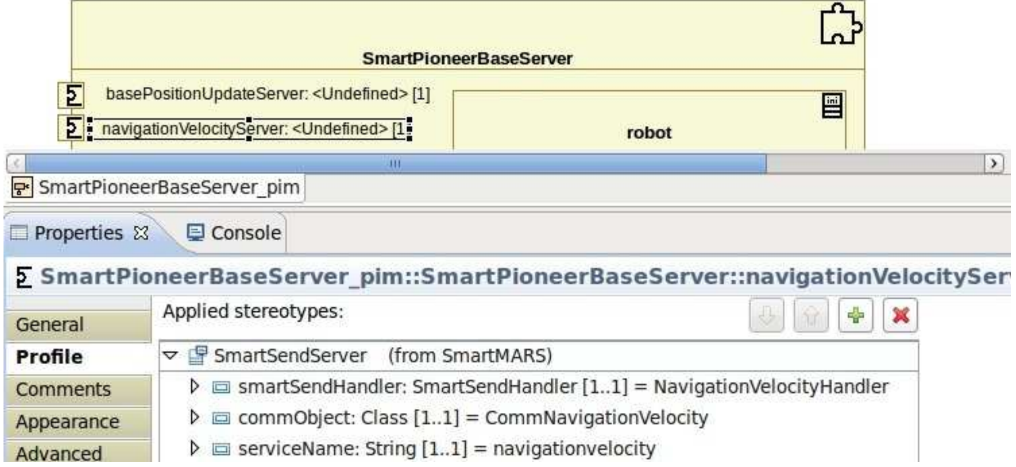


Figura 6.21: Puerto de comunicación navigationVelocityHandler

- *basePositionServer* → Puerto de comunicación de tipo *PushTimedServer* encargado de informar el estado actual del robot. Utiliza el objeto de comunicación *CommBaseState* y el servicio denominado *basestate*. El *Profile* de este puerto se muestra en la figura 6.22.

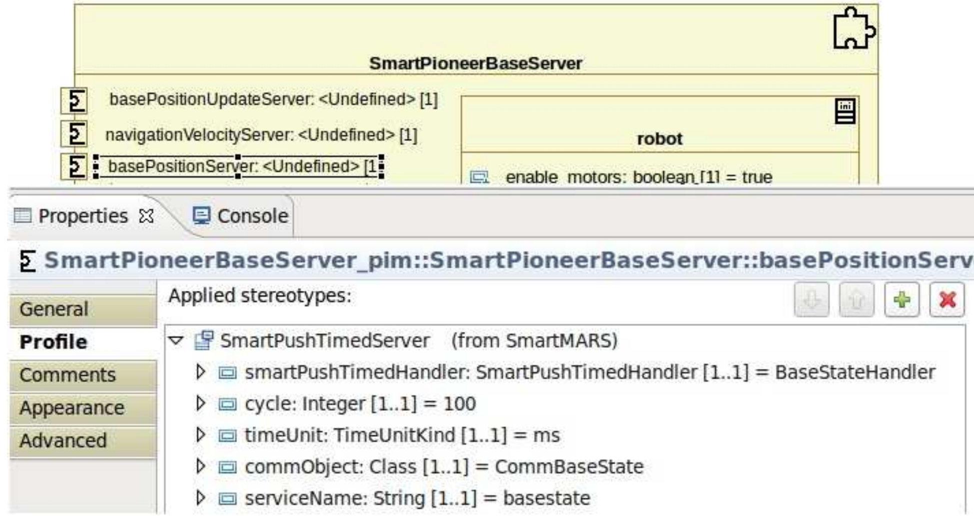


Figura 6.22: Puerto de comunicación basePositionServer

Por tanto, al ser la utilidad semejante, y coincidir el patrón, objeto y nombre del servicio en ambos casos con respecto a cómo eran los puertos del *SmartPlayerStageSimulator*, podemos concluir que no es necesario realizar cambio alguno en los componentes *SmartUPCTConsole* y *ExSmartGoTo* a la hora de intercambiar el componente simulador por el *SmartPioneerBaseServer*.

6.2.2. Construcción del deployment *DeployGoToRobot*

En este apartado se detalla el proceso de construcción del deployment donde relacionar los componentes *SmartUPCTConsole*, *ExSmartGoTo* y *SmartPioneerBaseServer.* Para ello podemos utilizar el deployment *DeployGoTo* que hemos diseñado en el apartado anterior, eliminando del modelo el componente *SmartPlayerStageSimulator* y reemplazándolo por el *SmartPioneerBaseServer*, pero estaremos creando una aplicación a costa de eliminar la anterior. Resulta por tanto interesante realizar cada aplicación en un deployment diferente.

Comenzamos creando el proyecto:

- *File / New / Project → Smart Robotic Project / SmartSoft Deployment →* Fijamos el nombre del componente: *DeployGoToRobot.*

A continuación abrimos el modelo:

- *DeployGoToRobot / model / pim / DeployGoToRobot _pim.di2*

El siguiente paso será importar al proyecto los componentes que van a formar parte del deployment:

- Desde la vista Outline: *Botón derecho sobre 'DeployGoToRobot' / Smart Robotics / Import Component. Seleccionamos los componentes SmartUPCTConsole*, *ExSmartGoTo* y *SmartPioneerBaseServer.*

A continuación desde la pestaña *Outline*, pinchamos sobre cada uno de los componentes y los arrastramos al deployment para hacerlos formar parte de él. Introducimos también el NamingService fijando una configuración local (*Ip = 0.0.0.0, Port = 12345).* Por último comunicamos los componentes a través de los puertos de comunicación mediante el *'Connection'* de la vista *'Palette'* del apartado *'SmartSoft Deployment'*. El modelo finalmente queda como aparece en la figura 6.23.

Por último realizamos el proceso de deploy sobre el proyecto:

- *Botón derecho sobre el proyecto / Smart Robotics / Deploy*

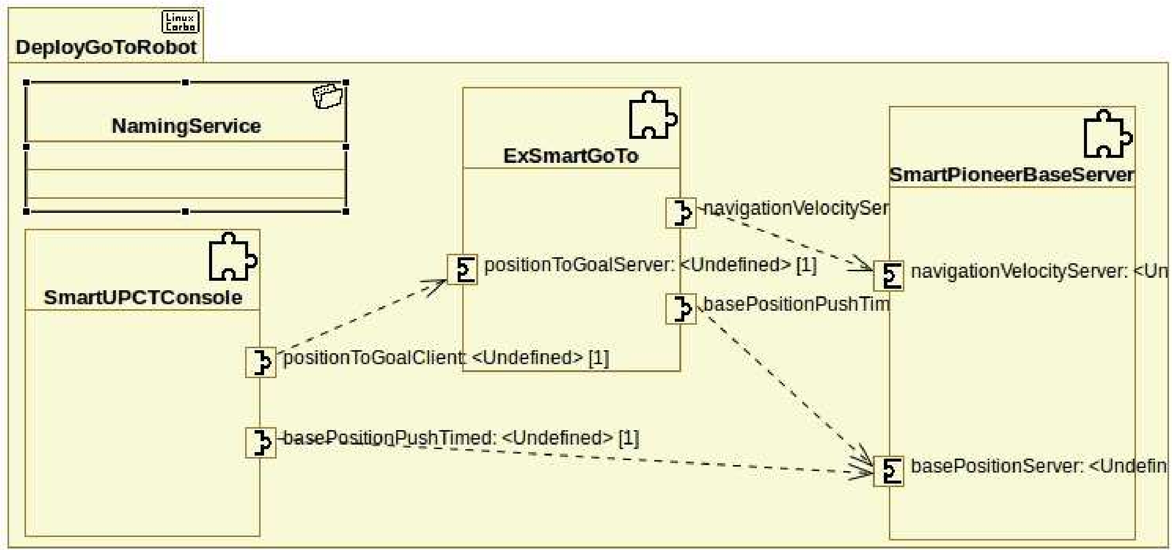


Figura 6.23: Modelo del deployment DeployGoToRobot

6.2.3. Ejecución de la aplicación en el robot real

La ejecución de esta aplicación en principio se hace de igual modo a como lo hemos ido haciendo hasta ahora. Sin embargo, en este apartado vamos a mostrar también cómo es posible ejecutar una aplicación en una máquina en la que no se dispone de la herramienta *SmartSoftMDSD*. Es decir, se dispone del entorno de ejecución que proporciona el framework, pero no así de la herramienta de desarrollo por modelos.

La razón de no tener instalada la herramienta de desarrollo en una máquina puede resultar interesante desde el punto de vista de ahorrar recursos del sistema. Quizás interese no tener instalado este tipo de software en el ordenador empotrado del robot real. Por tanto, el proceso sería desarrollar la aplicación en una máquina cualquiera en la que se dispone de la herramienta *SmartSoftMDSD*, y más tarde trasladar dicha aplicación al robot real lista para ser ejecutada (lógicamente el framework *CORBA/SmartSoft* debe estar instalado en el robot en cualquier caso).

Distinguimos entonces dos posibilidades: ejecutar la aplicación diseñada directamente en el robot real, o bien ejecutar la aplicación que ha sido trasladada desde una máquina remota donde se ha implementado.

Caso 1 – Aplicación diseñada directamente en el robot

En este caso, para ejecutar la aplicación seguiremos los pasos que utilizado hasta ahora. El proyecto asociado al deployment se encuentra dentro de la carpeta *ws-mdsd*. Para lanzar la ejecución, accedemos a la carpeta /src del proyecto e introducimos el siguiente comando.

```
smartsoft@smartsoft-vm:~/SOFTWARE/ws-mdsd/DeployGoToRobot/src$ ./deplyGoToTobot.sh start
```

Caso 2 – Aplicación diseñada en un ordenador remoto

Este ha sido el caso que hemos utilizado en la realización de este PFC, debido a un problema con la interfaz gráfica de usuario del sistema operativo instalado en el ordenador empotrado del robot *Pioneer P3-AT* del laboratorio (*Ubuntu 10.04 LTS*), con lo que no ha sido posible instalar en esta máquina la herramienta de desarrollo por modelos *SmartSoftMDSD*. Por otro lado, todo el framework de SmartSoft necesario para la ejecución de las aplicaciones se ha conseguido instalar siguiendo los pasos descritos en el *"anexo B – Instalación SmartSoft"*.

La solución adoptada ha sido la siguiente: realizar el desarrollo de los componentes y aplicaciones robóticas en otra máquina cualquiera en la que se disponga de la herramienta *SmartSoftMDSD* (ya sea instalándola o haciendo uso de la imagen de la máquina virtual), para más tarde llevar dicho desarrollo una vez compilado al robot y ejecutarlo. En este apartado se describe este proceso.

El proceso de desarrollo de una aplicación utilizando la herramienta *SmartSoftMDSD* se describió anteriormente en el apartado *"4.3 – Descripción del proceso de desarrollo"*. Partiendo desde este punto, la cuestión entonces es conocer cuáles son los cambios necesarios a realizar para que dicha aplicación o deployment podamos trasladarla y ejecutarla en el robot.

Los pasos necesarios para trasladar la aplicación desde un ordenador remoto al Pioneer P3-AT del laboratorio son los siguientes:

- ***Copiar los ejecutables de los componentes a la carpeta /src del deployment***
- ***Modificar el script asociado al deployment***
- ***Trasladar deployment***
- ***Trasladar nuevos objetos de comunicación utilizados por los componentes***
- ***Ejecución de la aplicación***

Copiar los ejecutables de los componentes a la carpeta src del deployment

Para llevar a cabo la ejecución de todo componente de SmartSoft en un deployment, únicamente es necesario el ejecutable de dicho componente y el fichero de inicialización *"*.ini"* asociado al mismo. El fichero de inicialización es generado por el proceso de *deploy* y está alojado en el directorio src del proyecto del deployment. Por otro lado como se describió en el *apartado 4.4 – Estructura de directorios de SmartSoft*, todos los ejecutables de los componentes se encuentran en la carpeta *SMART_ROOT/bin.* El script del deployment buscará en ese directorio el ejecutable del componente a lanzar.

Si queremos trasladar la aplicación completa desde un ordenador remoto hasta el *Pioneer*, resulta interesante realizarlo de la manera más compacta posible. El autor recomienda entonces, copiar los ejecutables de los componentes que vaya a utilizar el deployment, a la carpeta src del deployment, junto al fichero de inicialización. De esta manera para cada uno de los componentes tendremos la tupla *ejecutable-fichero ".ini"*.). La carpeta */src* del proyecto deployment queda finalmente de la siguiente manera:

```
smartsoft@smartsoft-vm:~/ws-mdsd/DeployGoToRobot/src$ ls
DeployGoToRobot_cheddar.xml  exSmartGoTo      smartPioneerBaseServer      smartUPCTConsole
deployGoToRobot.sh           ExSmartGoTo.ini  SmartPioneerBaseServer.ini  SmartUPCTConsole.ini
smartsoft@smartsoft-vm:~/ws-mdsd/DeployGoToRobot/src$
```

Modificar el script asociado al deployment

Cuando realizamos el proceso de "*deploy*" sobre un proyecto de tipo deployment se genera un script con el mismo nombre que el proyecto en la carpeta */src*, que se encarga de poner en marcha el *NamingService* y cada uno de los componentes que forman parte de él. Editando el fichero script denominado DeployGoToRobot.sh de la aplicación, observamos lo siguiente:

```
echo "starting components..."
xterm -e $SMART_ROOT/bin/exSmartGoTo -filename=ExSmartGoTo.ini -ORBInitRef
NameService=corbaloc:iiop:0.0.0.0:12345/NameService &
xterm -e $SMART_ROOT/bin/smartPioneerBaseServer -filename=SmartPioneerBaseServer.ini -ORBInitRef
NameService=corbaloc:iiop:0.0.0.0:12345/NameService &
xterm -e $SMART_ROOT/bin/smartUPCTConsole -filename=SmartUPCTConsole.ini -ORBInitRef
NameService=corbaloc:iiop:0.0.0.0:12345/NameService &
```

Vemos como cada componente se lanza en una terminal diferente mediante el comando *xterm*, sirviéndose de los ejecutables y de los ficheros de inicialización *"*.ini"*. Debido a que no se dispone de la interfaz gráfica en el ordenador empotrado del robot, el comando *xterm* no es posible ejecutarlo. La solución pasa por modificar estas instrucciones de la siguiente manera:

```
./smartPioneerBaseServer -filename=SmartPioneerBaseServer.ini -ORBInitRef
NameService=corbaloc:iiop:0.0.0.0:12345/NameService &
./exSmartGoTo -filename=ExSmartGoTo.ini -ORBInitRef
NameService=corbaloc:iiop:0.0.0.0:12345/NameService &
./smartUPCTConsole -filename=SmartUPCTConsole.ini -ORBInitRef
NameService=corbaloc:iiop:0.0.0.0:12345/NameService
```

Con estas instrucciones ejecutamos los componentes en segundo plano (un proceso por componente) sin utilizar el comando *xterm*, siendo lanzados desde la ubicación actual ya que en el punto anterior nos encargamos de modificar su ubicación.

Los ejecutables siempre se lanzan en primer plano por defecto y en segundo plano mediante el "&" final. Si existe algún componente en la aplicación que hace las veces de consola tomando datos por teclado, debemos tener la precaución de lanzarlo en último lugar (de otra manera no podremos lanzar el resto de componentes) y en primer plano (para poder interactuar con él). Por tanto lanzamos los ejecutables *smartPioneerBaseServer* y *exSmartGoTo* en segundo plano, y el componente consola *smartUPCTConsole* en primer plano.

Trasladar deployment

Una vez realizados los cambios en el script y copiados los ejecutables en la carpeta */src*, pasamos a trasladar el deployment al *Pioneer.* Para llevar a cabo esta tarea podemos adoptar diferentes soluciones. En este PFC se ha utilizado el comando *scp*, que permite enviar un fichero a una máquina remota a través de la red.

```
smartsoft@smartsoft-vm:~/SOFTWARE/ws-mdsd$ scp -r DeployGoToRobot p3at@192.168.2.132:/home/p3at
```

Los parámetros con los que ejecutamos el comando son: el fichero enviado (*DeployGoToRobot*), nombre de usuario en la máquina remota (*p3at*), dirección ip de la máquina remota obtenida mediante el comando *ifconfig* (*192.168.2.132*), y directorio en el que se alojará el fichero enviado (*/home/p3at*). Como el directorio que hemos enviado contiene todo lo necesario para la ejecución de la aplicación, no tenemos por qué preocuparnos de la ubicación exacta que le asignamos, pudiendo elegir aquella que más cómoda nos sea.

Una vez pulsada la tecla "*Enter*", se solicita la contraseña de usuario de la máquina remota antes de empezar a transmitir. En el caso del *Pioneer P3-AT* del laboratorio, dicha contraseña coincide con el nombre de usuario (*p3at*).

Trasladar nuevos objetos de comunicación utilizados por los componentes

A la hora de trasladar una aplicación debemos tener en cuenta los objetos de comunicación que utilizan los componentes en la aplicación. Este paso no será necesario si los objetos utilizados, forman parte de alguno de los repositorios que nos ofrece el equipo de SmartSoft (se proporcionan los objetos ya compilados) siempre y cuando no hayan sido modificados. Si hemos creado o modificado algún objeto de comunicación que más tarde utiliza algún componente en la aplicación a trasladar, debemos trasladar también el repositorio de objetos en el que este objeto se contiene.

En el caso concreto de nuestra aplicación recordemos que en el apartado "6.1.1 - Construcción del objeto de comunicación *CommObjectGoTo"* hemos diseñado un nuevo objeto de comunicación dentro del repositorio *CommUPCTObjects,* que es utilizado por el componente *SmartUPCTConsole* para enviar al *ExSmartGoTo* la posición y ángulo objetivo que el usuario ha introducido por teclado. Este objeto de comunicación debe ser también trasladado al robot real.

Para llevar a cabo esta tarea, únicamente tendremos que trasladar el fichero "*lib*.a"* que se encuentra ubicado en el directorio *$SMART_ROOT/lib* (con * igual al nombre del repositorio de objetos en el que se contiene el nuevo objeto); a la carpeta de idéntico nombre en la máquina del Pioneer P3-AT. Como en el punto anterior, utilizaremos el comando *scp.*

```
smartsoft@smartsoft-vm:~/SOFTWARE/smartsoft/lib$ ls -l
total 54664
-rw-r--r-- 1 smartsoft smartsoft 8002426 2011-12-12 19:10 libCommBasicObjects.a
-rw-r--r-- 1 smartsoft smartsoft 1593628 2011-09-20 09:21 libCommManipulationPlannerObjects.a
-rw-r--r-- 1 smartsoft smartsoft 5456214 2011-09-20 09:22 libCommManipulatorObjects.a
-rw-r--r-- 1 smartsoft smartsoft 5402458 2011-09-20 09:23 libCommNavigationObjects.a
-rw-r--r-- 1 smartsoft smartsoft 4209958 2011-09-20 09:24 libCommObjectRecognitionObjects.a
-rw-r--r-- 1 smartsoft smartsoft 2203904 2011-09-20 09:24 libCommPersonDetectionObjects.a
:
-rw-r--r-- 1 smartsoft smartsoft  258662 2011-12-22 16:41 libCommUPCTObjects.a
:
smartsoft@smartsoft-vm:~/SOFTWARE/smartsoft/lib$ scp -r libCommUPCTObjects.a
p3at@192.168.2.132:/home/p3at/SOFTWARE/smartsoft/lib
```

En este caso, si que tenemos que asegurarnos que el fichero trasladado se ubica concretamente en el mismo directorio en el robot real que en la máquina remota: *$SMART_ROOT/lib*.

Ejecución de la aplicación

Trabajando ya desde una consola en la máquina del *Pioneer*, encontramos el deployment en la ubicación que anteriormente hemos seleccionado (en concreto seleccionamos el directorio /home/p3at). En este directorio se encuentra todo lo necesario para llevar a cabo la ejecución de la aplicación, siendo posible recolocarla al gusto del usuario sin problema alguno.

Por último, para comenzar la ejecución de la aplicación lanzamos el script *deployGoToRobot* generado tal y como hemos ido haciendo hasta ahora. En la figura 6.24 se muestran los pasos necesarios desde el proceso de login.

```
Ubuntu 10.04.3 LTS p3at tty1

p3at login: p3at
Password:
Last login: Thu Jan 12 12:06:21 CET 2012 on tty2
Linux p3at 2.6.32-37-generic #81-Ubuntu SMP Fri Dec 2 20:35:14 UTC 2011 i686
/Linux
Ubuntu 10.04.3 LTS

Welcome to Ubuntu!
 * Documentation:  https://help.ubuntu.com/

51 packages can be updated.
0 updates are security updates.

p3at@p3at:~$ cd DeployGoToRobot/src/
p3at@p3at:~/DeployGoToRobot/src$ ls
DeployGoToRobot_cheddar.xml  ExSmartGoTo.ini             smartUPCTConsole
deployGoToRobot.sh           smartPioneerBaseServer      SmartUPCTConsole.ini
exSmartGoTo                  SmartPioneerBaseServer.ini
p3at@p3at:~/DeployGoToRobot/src$ ./deployGoToRobot.sh start
```

Figura 6.24: Ejecución de la aplicación en el robot Pioneer 3-AT - 1

Una vez pulsada la tecla *Enter*, comenzarán a lanzarse los distintos componentes que forman parte de la aplicación. Debido a que estamos trabajando con un robot real, el componente *SmartPioneerBaseServer* necesita un corto periodo de tiempo para conectarse y configurarse correctamente al software propio del robot. No es motivo de alarma entonces, que en el momento en el que el script lanza el componente *SmartPioneerBaseServer*, éste imprima por pantalla lo que aparece en la figura 6.25.

Tras ese periodo de tiempo (entre 5 y 10 segundos), el componente finalmente informará que se encuentra conectado correctamente al robot *Robosoft_3151 - Pioneer p3at-sh*.

```
Error reading packet header from robot connection: P2OSPacket():Receive():read()
:: Resource temporarily unavailable
Error reading packet header from robot connection: P2OSPacket():Receive():read()
:: Resource temporarily unavailable
Error reading packet header from robot connection: P2OSPacket():Receive():read()
:: Resource temporarily unavailable
Error reading packet header from robot connection: P2OSPacket():Receive():read()
:: Resource temporarily unavailable
Error reading packet header from robot connection: P2OSPacket():Receive():read()
:: Resource temporarily unavailable
turning off NONBLOCK mode...
Done. Connected to Robosoft_3151, a Pioneer p3at-sh, currbaud = 3
start...
connected.
connecting to: SmartPioneerBaseServer: basestate
connected.
```

Figura 6.25: Ejecución de la aplicación en el robot Pioneer 3-AT - 2

A continuación el componente SmartUPCTConsole imprimirá por pantalla el menú principal. Al igual que pasaba cuando ejecutábamos la aplicación utilizando el simulador, debemos utilizar la opción “23”. Llegado este punto, se solicita al usuario la posición y ángulo objetivo al que se desea que se desplace el robot tal y como aparece en la figura 6.26.

```
22 - SchunkGripper param
23 - ToGoal
98 - State Client
99 - Demos
00 - EXIT
--------------------------------------------------------------------------

please choose number:  23
Posicion [X,Y] actual: [0,0,0]
Enter the X coordinate Y coordinate (in milimeters) and Angle (in degrees) of th
e position you want to move the robot to (x y angle): 500 500 0
```

Figura 6.26: Ejecución de la aplicación en el robot Pioneer 3-AT - 3

A continuación se presentan unas imágenes donde se observa la trayectoria seguida por el robot Pioneer 3-AT en el laboratorio del DSIE. En la figura 6.27 el robot se encuentra aún en reposo en la que será su posición y orientación de partida (*x=0, y=0, ángulo=0º*). La posición objetivo es la [1000,1000] (expresamos las distancias en milímetros), por lo que el robot se desplazará un metro hacia adelante y un metro hacia su izquierda. Por otro lado la orientación objetivo siguen siendo los 0º actuales.

Figura 6.27: Movimiento del robot Pioneer 3-AT por el laboratorio - 1

Tras introducir la posición y ángulo objetivo el robot comenzará a moverse tal y como se aprecia en la figura 6.28.

Figura 6.28: Movimiento del robot Pioneer 3-AT por el laboratorio – 2

Finalmente en la imagen 6.29 vemos como el robot alcanza finalmente su destino. Llegado este punto, en la consola el usuario podrá elegir una nueva posición y ángulo objetivo o por el contrario dar por finalizada la aplicación.

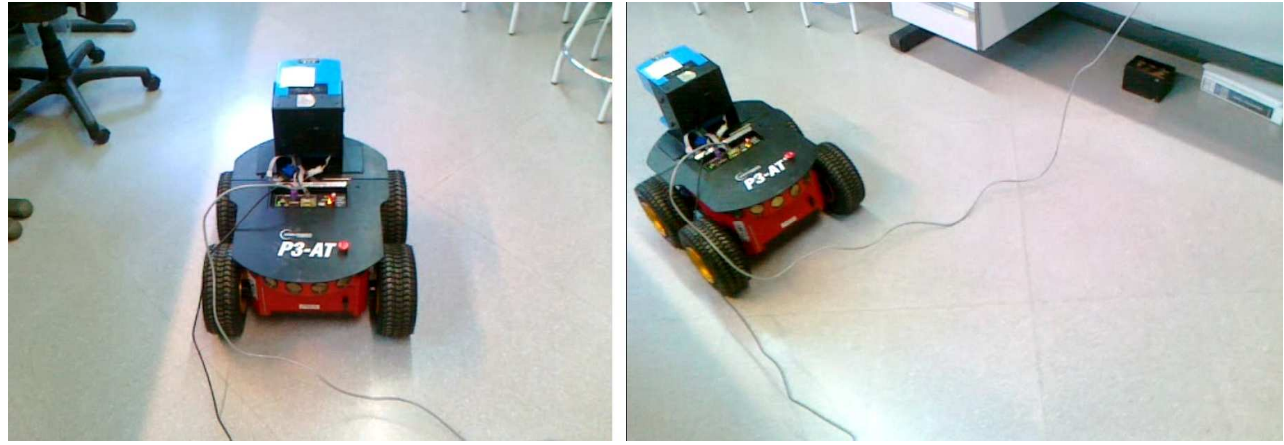


Figura 6.29: Movimiento del robot Pioneer 3-AT por el laboratorio – 3

7. CONCLUSIONES Y TRABAJOS FUTUROS

Trabajo realizado

Debido a la elevadísima complejidad de las aplicaciones robóticas actuales, cada vez se hace más común la utilización de frameworks y middlewares con los que conseguir facilitar el desarrollo de estas tareas. Primero se ha analizado el estado de la técnica en el campo de la Ingeniería de Software para la robótica, describiendo las distintas arquitecturas robóticas así como las posibles formas con las que podemos abordar su implementación: programación modular, frameworks orientados a objetos, frameworks orientados a componentes y desarrollo de software dirigido por modelos. En este aspecto, durante todo el PFC se ha puesto de manifiesto repetidamente las ventajas obtenidas al hacer uso de los recientes métodos de implementación de aplicaciones robóticas frente a los enfoques más tradicionales.

Una vez hecho esto, nos hemos centrado en SmartSoft analizando en detalle su funcionamiento. El equipo de desarrollo de SmartSoft ofrece un software que consiste en la unión de un framework orientado a componentes y una herramienta de desarrollo basada en modelos que simplifica enormemente las tareas de desarrollo que el usuario llevará a cabo más adelante al servirse de ésta.

Se ha analizado el toolchain describiendo cada una de las partes que lo forman (objetos, componentes, deployments, puertos, patrones…), describiendo el uso que hacen de éste los usuarios según sean sus roles (constructor de objetos de comunicación, constructor de componentes o generador de aplicaciones), y la manera en la que llevan a cabo su trabajo.

Finalmente hemos hecho uso de SmartSoft, incrementando poco a poco el nivel de complejidad de nuestros diseños, realizando desde pequeños ejemplos donde comprobar el funcionamiento de los distintos patrones de comunicación, hasta la implementación de una aplicación robótica de navegación tanto en el simulador PlayerStage como en el robot Pioneer P3AT por el laboratorio de DSIE (División de Sistemas e Ingeniería Electrónica) haciendo uso de este framework. Resulta de especial interés esta última aplicación ya que en su diseño se ha hecho un repaso sobre todos los pasos a llevar a cabo en el desarrollo de cualquier otra aplicación.

Para la realización de este PFC se ha utilizado documentación proporcionada por el equipo de desarrollo de SmartSoft resolviendo las deficiencias encontradas en ésta, y completado con el estudio que el autor ha hecho del framework y la herramienta. Por tanto la documentación en sí, es una aportación del trabajo realizado en este PFC.

Desde el comienzo de este PFC, el grupo de investigación del DSIE ha estado en contacto con el equipo de desarrollo de SmartSoft de la Hochschule (Alemania). Parte del equipo de SmartSoft visitó en el mes de octubre de 2011 la UPCT, donde el Prof. Dr. Christian Schlegel impartió el seminario *"Introduction into Model-Driven Software Development for Robotics"* al que asistió el autor del PFC, y que se encuentra dentro de las actividades organizadas en el Programa de Doctorado "Tecnologías de la Información y Comunicaciones". Durante esta visita también se realizó un *feedback* entre ambos grupos de investigación.

Conclusiones

La sencillez y transparencia con la que se realizan las interacciones entre los componentes es algo que el usuario agradecerá de forma notable, a la vez que queda patente desde el primer momento el elevado grado de reutilización de componentes que es posible alcanzar gracias a la herramienta de SmartSoft.

Una vez el usuario tiene claros los conocimientos necesarios para llevar a cabo el desarrollo de una aplicación por simple que esta sea, la construcción de nuevas y más complejas aplicaciones se convierte en algo mucho más asequible. Esto por el contrario, produce que el usuario debe invertir una gran cantidad de tiempo en conocer a fondo la herramienta para así poder aprovechar todo lo que aporta.

Depende por tanto del usuario tomar la decisión de si le compensa o no hacer uso de este software, invirtiendo recursos en el aprendizaje a costa de servirse de los beneficios que ofrece el software de cara a posteriores proyectos.

De cara a hacer más sencillo y llevadero este aprendizaje inicial, resulta de gran ayuda la imagen de la máquina virtual lista para trabajar que ofrece el equipo de SmartSoft, así como el workspace incluido *ws-examples* que ofrece sencillos ejemplos sobre el uso y funcionamiento de los patrones de comunicación de SmartSoft.

Sin embargo, se sigue echando en falta documentación que en algunos casos resulta clave para comprender como funciona realmente SmartSoft. Desde un primer momento no se realiza una clara distinción entre el framework orientado a componentes y herramienta de desarrollo, así como de las partes que constituyen cada una de ellas, con lo que un usuario que no esté familiarizado con este tipo de software se puede ver abrumado por términos como *CORBA, ACE, TAO, MDSD,* etc.

También se echa en falta una descripción detallada del proceso de construcción de cada una de las partes fundamentales que constituyen toda aplicación de SmartSoft: objetos, componentes y deployments. Aquí podemos incluir: la descripción de la *Palette* en cada uno de los casos, que ficheros son generados automáticamente por la herramienta, para qué sirven exactamente esos ficheros generados, como estructura SmartSoft los ficheros de los proyectos, etc. El equipo de SmartSoft únicamente proporciona documentación mediante un tutorial y *screencasts*, del proceso de construcción de deployments.

Tras finalizar este PFC, el autor se queda con la sensación de que el equipo de SmartSoft se ha centrado en el desarrollo de nuevas aplicaciones cada vez más complejas (algo completamente lógico) como son: la integración del *Kinect* de Microsoft, el control de brazos robóticos sobre un robot móvil, la identificación de rostros y voz, construcción de una pasarela a *ROS*, etc. Sin embargo considero que han dejado un poco de lado a los nuevos usuarios que deciden utilizar SmartSoft como herramienta software de desarrollo, al no proporcionarle toda la documentación que este va necesitar (que en gran parte de casos es algo puntual). Al usuario se le proporcionan documentación y herramientas con las que poder empezar a trabajar y realizar sus primeros deployments de manera muy rápida, con lo que SmartSoft “se vende rápido”, pero a la hora de dar el paso de ser el usuario el que desde cero sea capaz de realizar él por si solo una aplicación se echa en falta documentación.

Por último destacar otras dificultades encontradas como han sido la imposibilidad a la hora de instalar SmartSoft en un sistema operativo diferente al que el equipo utiliza (debiendo instalar la misma distribución finalmente), o los problemas que ha acarreado disponer de un modelo de robot Pioneer diferente con respecto al que posee el equipo de SmartSoft (P3AT frente a P3DX).

También se han encontrado dificultados a la hora de utilizar la información de los sonars que proporciona el Pioneer del laboratorio, ya que la lectura de dichos datos no fue contemplada por el equipo de SmartSoft a la hora de diseñar el componente SmartPioneerBaseServer. De igual modo, también ha existido una serie de problemas con respecto a la comunicación con el laser SICK LMS200 del que dispone se dispone en el laboratorio del DSIE, debido a una configuración diferente *robot-laser* con respecto a la que ha sido utilizado por el equipo de SmartSoft.

Trabajos futuros

De cara a trabajos futuros, considero que este PFC puede servir como guía a todo aquel que desee iniciarse en el desarrollo de una aplicación robótica móvil, ya que se han analizado las distintas arquitecturas robóticas, los requisitos funcionales y no funcionales que se les exigen, las posibilidades de llevar a cabo las implementaciones de éstas, los frameworks orientados a componentes y middlewares actuales en este campo, así como la descripción del moderno proceso de desarrollo basado en modelos.

Por supuesto será de gran utilidad para todo aquel que desee utilizar SmartSoft como herramienta de desarrollo en aplicaciones de robótica móvil.

Gracias a los capítulos *“3 - SmartSoft”* y *“4 - Primeros pasos con SmartSoft”*, el lector encontrará toda la documentación necesaria en el desarrollo de una aplicación utilizando este software. Por otro lado, a partir de la aplicación y de los pasos descritos en el capítulo *“6 – Desarrollo completo utilizando SmartSoft“*, el lector encontrará todas las situaciones posibles a las que se puede enfrentar en el desarrollo de otra aplicación cualquiera, complicándola a partir de ahí todo lo necesario hasta alcanzar sus objetivos.

Resulta de gran interés el que dicha aplicación haya sido diseñada utilizando tanto el simulador *PlayerStage* como el robot Pioneer 3-AT del que dispone el DSIE, mostrando los cambios a realizar de un desarrollo a otro. Si un grupo de investigación decide trabajar con SmartSoft y no posee el robot utilizado en este PFC, siempre podrá utilizar el simulador y más tarde adaptar el componente *BaseServer* con el fin de modelar el robot del que dispongan.

Como trabajos futuros también hay que mencionar la inclusión y utilización de la información proporcionada tanto por los sonars como por el láser. Tras la realización de este PFC se le propondrá al equipo de SmartSoft si están interesados en alguna parte de éste, para traducir al inglés dichas partes y que ellos las utilicen en su web.

8. BIBLIOGRAFÍA

- *Atkinson (2003)*
  C. Atkinson and T. Kühne, "Model-driven development: a metamodelling foundation", IEEE Softw., vol. 20, no. 5, pp. 14–18, 2003. 1

- *Avitrack (2004)*
  Task 5.1 Framework Prototype. http://www.aero-scratch.net/avitrack.html

- *Arkin (1998)*
  Behavior-Based Robotics, Ronald C. Arkin, 1998

- *Brooks (2005)*
  Brooks, A.; Kaupp, T.; Makarenko, A.; Williams, S. & Orebäck, A. (2005). Towards Component-Based Robotics. Proceedings IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS), Canada

- *Brugali (2005)*
  Brugali, D. (2005). Principles and Practice of Software Development in Robotics (Workshop ICRA 2005), http://robotics.unibg.it/sdir2005/program.html

- *Brugali (2000)*
  Brugali, D. and G. Menga (2000). Frameworks and pattern languages: an intriguing relationship. ACM ComputingSurveys 32(2), 1-6.

- *Bruegge (1993)*
  Bruegge, B., Gottschalk, T., y Luo, B. (1993). A Framework for Dynamic Program Analyzers. ACM SIGPLAN Notices, 28:65–82.

- *Cañas (2004)*
  Cañas, J.M. Programación de robots móviles. Universidad Rey Juan Carlos, 2004.

- *Fayad y Schmidt (1997)*
  Fayad, M. E. y Schmidt, D. C. (1997). Object-Oriented Application Frameworks. Communications of the ACM, 40(10):32–38.

- *Gamma (1995)*
  Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns : Elements of Reusable Object Oriented Software. Addison-Wesley Pub Co, 1st edition, January 1995.

- *Goodwin (2008)*
  R. Simmons, J. Fernandez, R. Goodwin, S. Koenig, J. O'Sullivan: Lessons Learned From Xavier, IEEE Robot. Autom. Mag. 7(2), 33–39 (2008)

- *Heineman & Councill (2001)*
  Heineman, G. T. & Councill, W. T. (2001). Component based software engineering: putting the pieces together. Addison-Wesley

- *Hüni (1997)*
  Hüni, H., Johnson, R., y Engel, R. (1997). A Framework for Network Protocol Software. ACM SIGPLAN Notices, 32.

- *Iborra (2008)*
  Iborra, D. Alonso, F. J. Ortiz, J.A. Pastor, P. Sánchez, B. Álvarez. Experiences using Software Engineering in the design of Service Robots. RAM_2008.

- *Iborra (2009)*
  Iborra, D. Alonso, F. J. Ortiz, J.A. Pastor, P. Sánchez, B. Álvarez. Design of Service Robots. Experiences using Software Engineering in the design of Service Robots. RAM_2009.

- *Konolige (1997)*
  Konolige, K.; Myers, K. L.; Ruspini, E. H. & Saffiotti, A. (1997). The Saphira architecture: a design for autonomy. Journal of experimental & theoretical artificial intelligence (JETAI), 9(1):215-235

- *Kortenkamp (1998)*
  Kortenkamp, P. Simmons, M. Robotic System Architectures and Programming.

- *Lau (2007)*
  Lau, K. and Z. Wang (2007). Software componentes models. IEEE Trans. Software Eng.33(10), 709-724.

- *Mattsson (1999)*
  Mattsson, M. Object-Oriented Frameworks: A Survey of Methodological Issues. Technical

- *Mallet (2002)*
  Mallet, A.; Fleury, S. & Bruyninckx, H. (2002). A specification of generic robotics software components: future evolutions of GenoM on the OROCOS context. Proceedings IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS), pp. 2292-2297, Switzerland

- *Murphy (2000)*
  Introduction to AI Robotics, Robin R. Murphy, 2000

- *Montemerlo (2003)*
  Montemerlo, M.; Roy, N. & Thrun, S. (2003). Perspectives on standardization in mobile robot programming: the Carnegie Mellon navigation toolkit. Proceedings IEEE/RSJ Int. Conference on Intelligent Robots and Systems (IROS), pp. 2436-2441, Las Vegas Schlegel, C. & Wörz, R. (1999). The software framework

- *Parker (2008)*
  Parker, P. History of Intelligent Mobile Robotics, August-2008.

- *Nierstrasz (1995)*
  Nierstrasz, O. (1995). Requirements for a Composition Language. En Ciancarini, P., Nierstrasz, O., y Yonezawa, A. (eds.), Proc. of the ECOOP'94 Workshop on Object-Based Models and Languages for Concurrent Systems, n´um. 924 de LNCS, p´ags. 147–161. Springer-Verlag

- *Sánchez (2005)*
  Sánchez, Pastor, Alonso, Álvarez. Soporte a la Distribución para un Framework Basado en Componentes. IV Jornadas de introducción a la Investigación de la UPCT.

- *Schlegel y Wörz (1999)*
  Schlegel, C. & Wörz, R. (1999). The software framework SmartSoft for implementing sensorimotor systems. Proceedings IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS), pp. 1610-1616, Korea

- *Schlegel (2004)*
  Schlegel, C. (2004). Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach. PhD thesis, Faculty of Computer Science, Univ. of Ulm, http://www.rz.fh-ulm.de/~cschlege

- *Schlegel (2006)*
  Christian Schlegel. Communication Patterns as Key Towards Component-Based Robotics. International Journal on Advanced Robotics Systems, Special Issue on Software Development and Integration in Robotics, 3(1):49-54, 03 2006.

- *Schlegel (2009)*
  Christian Schlegel, Thomas Haßler, Alex Lotz and Andreas Steck. Robotic Software Systems: From Code-Driven to Model-Driven Designs. In Proc. 14th Int. Conf. on Advanced Robotics (ICAR), Munich, 2009.

- *Schlegel (2010)*
  Christian Schlegel, Andreas Steck. Model-Driven Software Development in Robotics. IROS 2010 Workshop (W13, October 22).

- *Schmidt (1998)*
  M.Schmidt. "The design of the TAO real-time object request broker". Computer Communications Special Issue on Building Quality of Service into Distributed Systems, 21(4), April 1998.

- *Shaw (2006)*
  Shaw, M. and P. Clements (2006). The golden age of software architecture. IEEE Softw. 23(2), 31-39.

- *Siegwart (1998)*
  Siegwart, R. Nourbakhsh, I. Introduction to Autonomous Mobile Robots. Intelligent Robotics and Autonomous Agents. Ronald C. Arkin, editor

- *SmartSoft (2005)*
  SmartSoft. (2005). Reference Implementation. http://smart-robotics.sourceforge.net/

- *Szyperski y Pfister (1997)*
  Szyperski, C. y Pfister, C. (1997). Summary of the Workshop on Component Oriented Programming (WCOP'96). En M¨uhlh¨auser, M. (ed.), Special Issues in Object-Oriented Programming. Workshop Reader of ECOOP'96. Dpunkt Verlag.

- *Szyperski (1998)*
  C. Szyperski. Component Software. Beyond Object-Oriented Programming.Addison-Wesley Longman, 1998.

- *Taylor (1996)*
  Taylor, R. et al. (1996). A Component- and Message-Based Architectural Style for GUI Software. IEEE Transactions on Software Engineering, 22(6):390–406.

- *TINA (1995)*
  TINA-C (1995). Overall Concepts and Principles of TINA. (http://www.tinac.com/95/file.list.html).

- *Utz (2002)*
  Utz, H.; Sablatnög, S.; Enderle, S. & Kraetzschmar, G. (2002). MIRO – Middleware for mobile robot applications. IEEE Transactions on Robotics and Automation, 18(4):493-497

- *Vallecillo (2006)*
  Manuel F. Bertoa y Antonio Vallecillo. IEEE Members. Medidas de Usabilidad de Componentes Software

- *Vaughan (2003)*
  Vaughan, R.; Gerkey, B. & Howard, A. (2003). On device abstractions for portable, reusable robot code. Proceedings IEEE/RSJ IROS, pp. 2121-2427, Las Vegas

- *Vicente-Chicote (2010)*
  V3CMM: a 3-View Component Meta-Model for Model-Driven Robotic Software Development. Alonso, Vicente-Chicote, Ortiz, Pastor, Álvarez.

Anexo A - Puesta en marcha de la Imagen VMWare

A.1 Objetivo

Puesta en marcha de SmartSoft haciendo uso de la imagen VMWare proporcionada. La aplicación encargada de la virtualización será VMWare Player.

A.2 Software y versiones

A día de hoy (27/11/11), el software utilizado y sus versiones son:

- **VMWare Player v3.1.4**
  Producto gratuito que permite correr máquinas virtuales. Las máquinas virtuales se pueden crear con productos más avanzados como VMWare Workstation, GSX Server, ESX Server, o con el propio VMWare Player desde su versión 3.0 (las versiones anteriores no incluyen dicha funcionalidad). Encontramos la aplicación en la siguiente dirección: http://downloads.vmware.com/d/info/desktop_downloads/vmware_player/3_0
- **SmartSoft-ubuntu-VM_rev46+mdsd0.10.4.**
  La imagen de VMWare ofrece una versión instalada de SmartSoft con el entorno necesario, basado en Ubuntu LTS 10.04.2. La imagen VMWare incluye:
  - SmartSoft basado en CORBA
  - SmartSoft MDSD (Model Driven Software Development) Toolchain.

  La imagen VMWare de SmartSoft junto con el simulador *Player Stage* se encuentra disponible en la siguiente dirección: http://sourceforge.net/projects/smart-robotics/files/

A.3 Requisitos hardware

Los requisitos del sistema para poder trabajar con la imagen VMWare son:

- Versión de VMWare Player relativamente reciente (>= 3.1.3).
- > 2GB RAM
- 15 GB libre en el disco duro.

A.4 Proceso

Una vez descargados los ficheros necesarios e instalado correctamente VMWare Player, pasamos a descomprimir el fichero de la imagen introduciendo "*tar -xjvf <filename>*" en caso de utilizar Linux, o *"botón derecho / extraer aquí"* en caso de utilizar WinRar o 7zip en Windows.

Para empezar a trabajar con la máquina virtual seguimos los siguientes pasos:

- Iniciar el fichero SmartSoft-Virtual-Machine.vmx obtenido en VMWare Player.
  -- Linux:
    - Shell: *>> wmplayer SmartSoft-Virtual-Machine.vmx*
    - *GUI:*
      # arrancar VMWare Player
      # "file" / "open a virtual machine" y seleccionar el fichero anterior.
  - Windows: doble clic sobre el fichero anterior.
- Tras esto, VMWare Player iniciará la máquina virtual.
  - La máquina virtual aparece como un ordenador virtual dentro de su equipo.
  - Si se trata de la primera vez que arranca la imagen, VMWare le preguntará con el fin de configurar ciertas funciones de administración de red, si el usuario ha movido la máquina virtual, o por el contrario la ha copiado. Siempre seleccionamos la opción "I copied it".
  - En nuestro caso, este equipo virtual está a cargo de Ubuntu.
  - Se proporciona una instalación "lista-para-usar" del toolchain de SmartSoft.

- La información del sistema de la imagen virtual es la siguiente:
  - Imagen VMWare basada en Ubuntu LTS 10.04.2
  - Nombre de usuario / Password → *SmartSoft / SmartSoft*
  - Contraseña root en caso de ser necesaria → *SmartSoft*
- Siempre deberá suspenderse o apagarse la máquina virtual tal y como se realiza en los equipos normalmente.

En la imagen VMWare el framework de SmartSoft se instala en */ home / SmartSoft / SOFTWARE / SmartSoft*. Por otro lado, la variable de entorno *$SMART_ROOT* queda asociada con */ home / SmartSoft.*

Comentar que siempre será posible una comprobación SVN del repositorio de SourceForge. Para obtener la última versión de SmartSoft usar svn update.

```
>> cd $SMART_ROOT
>> svn update
>> cd src
>> make distclean; make
```

Anexo B – Instalación SmartSoft

B.1 Objetivo

Descripción del proceso de instalación de SmartSoft. Se especifican los paquetes adicionales necesarios, así como el proceso de instalación de ACE y TAO.

B.2 Software y versiones

A día de hoy (27/11/11), el software utilizado y sus versiones son:

- Ubuntu LTS 10.04.2.
Este es el sistema operativo elegido que tendrá la máquina donde se realice el proceso de instalación de SmartSoft. Es posible utilizar otros sistemas operativos como: Linux SuSE 10.3 – 11.2 – 11.3, Debian Lenny, Squeeze… En el computador empotrado del robot Pioneer P3-AT se intentó realizar la instalación sobre Ubuntu 8.04 dando lugar a una instalación bastante más tediosa. La elección de esta distribución en concreto es debido a que esta es la elegida por el equipo de desarrollo de SmartSoft, lo cual hace conveniente su uso. Podemos encontrar este sistema operativo en la siguiente dirección: http://releases.ubuntu.com/lucid/

- ACE+TAO-6.0.2
Se trata de la versión más reciente de ACE+TAO y será la elegida para la instalación. Aún así como veremos más adelante, el script de instalación buscará y descargará automáticamente la última versión disponible de este software.

- SmartSoft MDSD Toolchain 0.10.4
Se trata de la versión más reciente de SmartSoft. Al igual que ocurre con ACE+TAO, en el proceso de instalación se chequeará el repositorio del equipo de SmartSoft en busca de la última actualización del software.

B.3 Paquetes adicionales necesarios

Antes de realizar la instalación de ACE/TAO y SmartSoft, será necesario disponer en nuestra máquina de unos determinados paquetes. A continuación aparece de que paquetes se trata así como su utilidad, dejando para el apartado B.5 la descripción del proceso de instalación de estos:

- Build-essential
Metapaquete (paquete con referencias a otros paquetes) que contiene las instrucciones necesarias para instalar los paquetes esenciales para *programar en C/C++*.

- Doxygen
Contiene un sistema de documentación para C++, C, Java, Corba IDL... Útil para generar documentación HTML y/o un manual de referencia a partir de un grupo de ficheros fuente documentados.

- OpenCV
Open Source Computer Vision. Librería de visión artificial de código abierto originalmente desarrollada por Intel

- Boost
Archivos de desarrollo de las bibliotecas Boost C++

- Subversion
Sistema de Control de Versiones de código abierto.

- Graphviz
Conjunto de herramientas de dibujo y gráficos.

B.4 Requisitos hardware

- Algo más de 5 GB de espacio libres.

B.5 Proceso

1. Instalación de los paquetes extra.

```
>> sudo apt-get install build-essential doxygen-latex libhighgui-dev libcv-dev
liblapack-dev libboost-all-dev libcvaux-dev subversion libgraphviz-dev
```

2. A continuación descargamos el script que proporciona el equipo de SmartSoft para la instalación de ACE+TAO. Este script se encargará de todo el proceso: descarga de la última versión, extraer los ficheros, crear la estructura de directorios idónea, compilar las distintas partes de ACE+TAO... Este script lo encontramos en la siguiente dirección: http://smart-robotics.sourceforge.net/INSTALL-ACE+TAO-6.0.2.sh. Lo descargaremos en el directorio home del usuario (/home/<username>). A continuación le damos permiso de ejecución y lo lanzamos:

```
>> cd
>> wget http://smart-robotics.sourceforge.net/INSTALL-ACE+TAO-6.0.2.sh
>> chmod +x INSTALL-ACE+TAO-6.0.2.sh
>> sudo ./INSTALL-ACE+TAO-6.0.2.sh
```

3. El script justo antes de finalizar su ejecución, nos advierte que es necesario añadir unas determinadas variables de entorno. Lo más conveniente será introducirlas en el "/home/<username>/.profile", ya que si creamos las variables de entorno en línea se perderán entre sesiones, mientras que al ponerlas en el /.profile sabemos que estarán correctamente configuradas ya que éste siempre se ejecuta cada vez que nos logueamos. Para ello ejecutamos lo siguiente:

```
>> gedit /home/<username>/.profile
```

   Una vez abierto se añade al final del fichero las siguientes líneas:

```
export SMART_ROOT=/home/<usuario>/SOFTWARE/smartsoft
export ACE_ROOT=/opt/ACE_wrappers
export TAO_ROOT=/opt/ACE_wrappers/TAO
export LD_LIBRARY_PATH=$ACE_ROOT/lib
```

   Tener la precaución de dejar una línea en blanco (introducir un retorno de carro) al final del fichero.

4. Si queremos que los cambios tengan efecto y conseguir así cargar las variables de entorno ejecutamos la siguiente orden:

```
>> source /home/<usuario>/.profile
```

   Esto tendrá las mismas consecuencias que reiniciar la máquina y volver a loguearse.

5. El siguiente paso es descargar la última versión estable de SmartSoft de su repositorio mediante Subversion. El directorio donde descargar el archivo es el indicado por la variable SMART_ROOT.

```
>> cd $SMART_ROOT
>> svn co https://smart-robotics.svn.sourceforge.net/svnroot/smart-
robotics/trunk/smartsoft/ .
```

6. A continuación se lanza el Makefile encargado de compilar SmartSoft que se encuentra en la carpeta $SMART_ROOT/src.

```
>> cd $SMART_ROOT/src
>> make
```

7. Por último configuramos el Naming Service de CORBA. Será necesario volver al .profile del usuario e introducir una nueva variable de entorno:

```
>> gedit /home/<username>/.profile
```

   Y dentro del fichero:

```
export NameServiceIOR=corbaloc:iiop:localhost:12345/NameService
```

Anexo C - Manual de usuario del robot Pioneer 3-AT

C.1 Objetivo

Descripción del proceso de puesta en marcha y manual de funcionamiento del robot PioneerP3-AT del laboratorio del DSIE.

C.2 Plataforma robótica y versiones

Pioneer es una familia de robots móviles, tanto de 2 como 4 ruedas, que incluye modelos como: el Pioneer 1, Pioneer AT, Pioneer 2-DX, -DXe, -DXf, -CE, -AT, Pioneer 2-DX8/Dx8 Plus y -AT8/AT8 Plus, y el reciente Pioneer 3-DX y -AT.

Estas pequeñas plataformas de desarrollo e investigación comparten una arquitectura y un software de base común con el resto de plataformas de *MobileRobots*, incluyendo *AmigoBot, PeopleBot V1, Performance PeopleBot* y *PowerBot*. En la figura c.1 se muestran dos imágenes del robot Pioneer 3-AT y el la figura c.2 se muestra una composición con algunos de los componentes de la familia robótica Pioneer.

Figura c.1: Robot Pioneer 3-AT

Estos robots han sido empleados en un gran número de proyectos de investigación entre los que se encuentran algunos proyectos de la Agencia de Investigación y Proyectos Avanzados de Defensa (siglas inglesas *DARPA*). Su éxito se debe a que contienen todos los elementos básicos necesarios para sensorización y navegación, en un entorno real.

Todas las plataformas *MobileRobots* operan como servidores en un entorno cliente-servidor: su microcontrolador maneja todos los detalles de bajo nivel de la robótica móvil, manteniendo la velocidad y rumbo de la plataforma, la adquisición de la lectura de los sensores y el control de diversos accesorios, como un brazo robótico.

Las plataformas *MobileRobots* requieren la conexión de un PC en el que se ejecutará el software cliente que aporte la inteligencia al control del robot. Sin embargo, el robot Pioneer 3-AT del laboratorio contiene un ordenador empotrado, por lo que se trata de es un robot totalmente autónomo. A diferencia de otros robots comerciales de mayores dimensiones, el modesto tamaño del robot Pioneer 3-AT le permite una navegación óptima en espacios cerrados y pequeños, como habitaciones, laboratorios y pequeñas oficinas.

Figura c.2: Familia de robots Pioneer

El manual para el robot Pioneer 3-AT (P3AT) proporcionado por la empresa fabricante *MobileRobots* lo encontramos en la siguiente dirección: http://www.mobilerobots.com/ResearchRobots/P3AT.aspx

C.3 Puesta en marcha y funcionamiento del robot

El robot Pioneer del laboratorio como se ha comentado anteriormente posee un computador empotrado. En la parte superior del robot encontramos las entradas y salidas que posee el microcontrolador propio del Pioneer, así como la máquina empotrada. En la figura c.3 se muestra dicho panel con el robot apagado, donde distinguimos el panel de la máquina empotrada a la izquierda de la imagen, y el del microcontrolador del robot a la derecha.

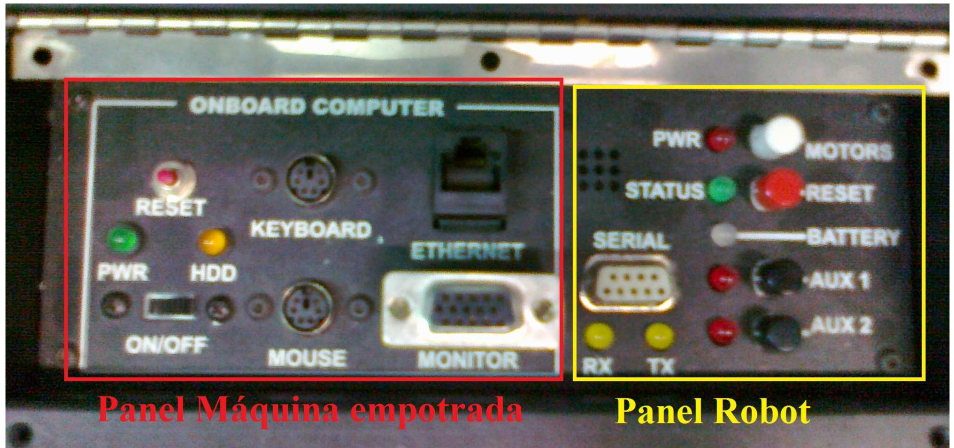


Figura c.3: Entradas y salidas del robot Pioneer 3-AT del laboratorio

Lo primero para comenzar a trabajar con el robot es conectarlo a través del interruptor que existe en el lateral trasero del robot que se muestra en la figura c.4. El robot dispone de una serie de baterías para poder trabajar de manera autónoma. Para cargar dichas baterías será suficiente con alimentar el robot a través del conector que también aparece en la parte inferior de la figura c.4. A la hora estar programando o trabajando sobre el robot, puede resultar conveniente quitar las baterías y alimentar el robot directamente a través de la red eléctrica.

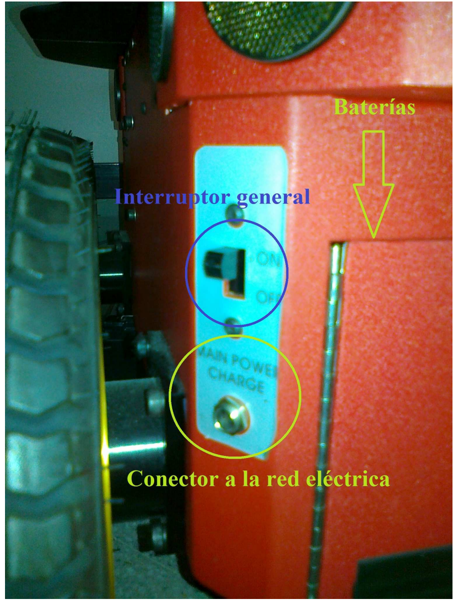


Figura c.4: Interruptor general del robot

Una vez coloquemos el interruptor en la posición ON, el robot arrancará. Si todo es correcto, se iluminarán los LED PWR y STATUS del panel de control del microcontrolador. Si por el contrario en el proceso de arranque el robot comienza a emitir un pitido pueden estar sucediendo 2 cosas:

- Que el robot no esté conectado a la red eléctrica y el nivel de batería es muy bajo. En este caso el LED BATTERY parpadeará.
- Que la seta de emergencia de la parte superior se encuentra retenida.

Pasamos entonces a arrancar el ordenador empotrado. Para ello debemos poner en ON el interruptor que aparece en el panel de la parte superior de la plataforma como vemos en la figura c.5. Una vez hecho esto, debe encenderse el LED PWR cercano al interruptor. De todas las entradas y salidas que posee el empotrado, existen 3 que deben estar forzosamente conectadas para poder trabajar con SmartSoft desde dicha máquina. Estas son:

- Teclado
- Monitor
- Cable de red con conexión a internet

Dicho panel queda entonces como se muestra en la figura c.5:

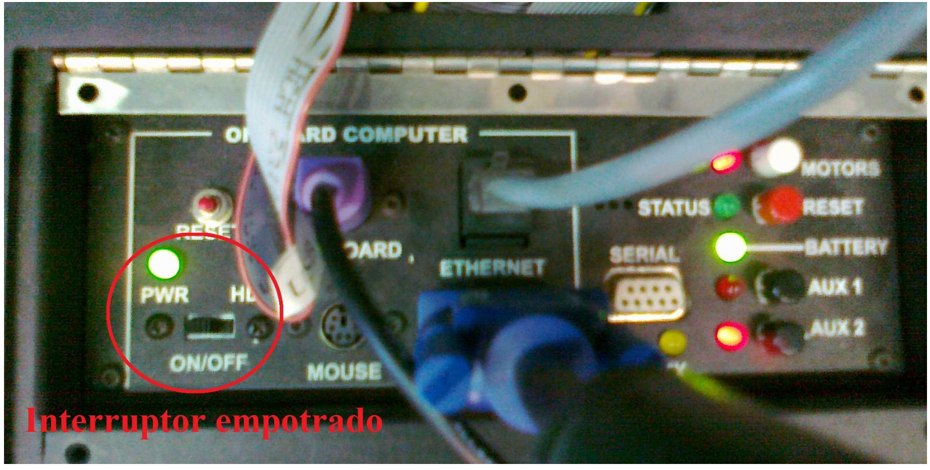


Figura c.5: Panel de control con el empotrado encendido

Una vez realizados estos pasos comenzará el arranque del sistema de la máquina empotrada. En el GRUB (gestor de arranque múltiple) aparecerá como opción por defecto la partición que contiene el sistema operativo *Ubuntu 10.04 LTS*, que ha sido instalado durante la realización de este PFC.

En dicha partición se encuentra instalado el framework de SmartSoft siguiendo los pasos descritos en el *"anexo B – Instalación SmartSoft"*. Sin embargo, debido a un problema con la interfaz gráfica de usuario del sistema operativo, no ha sido posible instalar en esta máquina la herramienta de desarrollo por modelos *SmartSoftMDSD*.

El usuario y contraseña con el que se instaló el sistema operativo son:

- *Usuario* → *p3at*
- *Contraseña* → *p3at*

Una vez realizado el proceso de login correctamente aparecerá algo similar a la figura c.6:

```
Ubuntu 10.04.3 LTS p3at tty1

p3at login: p3at
Password:
Last login: Thu Jan 12 12:06:21 CET 2012 on tty2
Linux p3at 2.6.32-37-generic #81-Ubuntu SMP Fri Dec 2 20:35:14 UTC 2011 i686
/Linux
Ubuntu 10.04.3 LTS

Welcome to Ubuntu!
 * Documentation:  https://help.ubuntu.com/

51 packages can be updated.
0 updates are security updates.

p3at@p3at:~$ cd DeployGoToRobot/src/
p3at@p3at:~/DeployGoToRobot/src$ ls
DeployGoToRobot_cheddar.xml  ExSmartGoTo.ini              smartUPCTConsole
deployGoToRobot.sh           smartPioneerBaseServer       SmartUPCTConsole.ini
exSmartGoTo                  SmartPioneerBaseServer.ini
p3at@p3at:~/DeployGoToRobot/src$ ./deployGoToRobot.sh start
```

Figura c.6: Consola en robot Pioneer 3-AT

Trabajando ya desde la consola en la máquina del *Pioneer*, podemos ejecutar cualquier deployment accediendo a la carpeta /src del proyecto y lanzando el script asociado de la siguiente manera:

```
p3at@p3at:~/cd DeployGoToRobot/src$ ./deployGoToRobot start
```