ESCUELA TÉCNICA SUPERIOR DE
INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD DE MÁLAGA

PROYECTO FIN DE CARRERA

DESARROLLO DE UN SERVICIO DE VIDEOJUEGOS CONVERSACIONALES MULTIJUGADOR EN RED PARA DISPOSITIVOS MÓVILES

INGENIERÍA DE TELECOMUNICACIÓN

MÁLAGA, 2010 MARCOS TOLEDO PARIENTE

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD DE MÁLAGA

Titulación: Ingeniería de Telecomunicación

Reunido el tribunal examinador en el día de la fecha, constituido por:

D./Dª. _____

D./Dª. _____

D./Dª. _____

para juzgar el Proyecto Fin de Carrera titulado:

DESARROLLO DE UN SERVICIO DE VIDEOJUEGOS CONVERSACIONALES MULTIJUGADOR EN RED PARA DISPOSITIVOS MÓVILES

del alumno/a D./Dª. *Marcos Toledo Pariente*

dirigido por D./Dª. *Francisco Javier González Cañete*

ACORDÓ POR _____ OTORGAR LA CALIFICACIÓN DE _____

Y, para que conste, se extiende firmada por los componentes del tribunal, la presente diligencia

Málaga, a _____ de _____ de _____

El/La Presidente/a El/La Vocal El/La Secretario/a

Fdo.: _____ Fdo.: _____ Fdo.: _____

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD DE MÁLAGA

DESARROLLO DE UN SERVICIO DE VIDEOJUEGOS CONVERSACIONALES MULTIJUGADOR EN RED PARA DISPOSITIVOS MÓVILES

REALIZADO POR:

Marcos Toledo Pariente

DIRIGIDO POR:

Francisco Javier González Cañete

DEPARTAMENTO DE: Tecnología Electrónica

TITULACIÓN: Ingeniería de Telecomunicación

PALABRAS CLAVE: motor videojuegos conversacionales multijugador, J2ME, Java, XML, arquitectura MVC.

RESUMEN:

Debido al auge que está experimentando el sector del ocio interactivo, así como a la creciente implantación de terminales móviles con mayores prestaciones y numerosas posibilidades de conectividad, se plantea el presente proyecto, cuyo objetivo es desarrollar una plataforma de videojuegos conversacionales orientada a dispositivos móviles que permita a varios jugadores participar de forma simultánea en la misma partida. Se ha desarrollado un motor basado en Java sobre el que se pueden ejecutar este tipo de videojuegos, que permite la sencilla incorporación de nuevos títulos por medio de la creación de unos archivos de configuración XML en los que se incluyen las localizaciones, objetos y acciones que se pueden realizar en la aventura. La aplicación que reside en los terminales móviles se ha realizado en base a la tecnología J2ME, y se comunica con el servidor por medio de un protocolo de creación propia.

Málaga, Enero de 2010

Agradecimientos

Me gustaría aprovechar estas líneas para humildemente dar las gracias a toda la gente que me ha ayudado y servido de apoyo a lo largo de este período que toca a su fin.

En primer lugar agradecer a mi tutor Francisco Javier González Cañete su gran dedicación y detallado seguimiento a lo largo de la elaboración del proyecto, así como su aporte de ideas y soluciones ante los problemas que me iban surgiendo.

A mis amigos David, Isabel y Mario por su colaboración desinteresada y por la paciencia que han demostrado conmigo.

A mis compañeros de estudios Alfonso, Bernardo, José, Juan Carlos, Paco, Pepe, Salva y Simón, a los que tengo la suerte de poder llamar amigos. Guardo con cariño los buenos momentos que me han hecho pasar durante el transcurso de la carrera.

Finalmente, a mi familia, y en especial a mis padres por su apoyo incondicional y por ser mis mayores valedores.

A todos vosotros, gracias.

ÍNDICE DE CONTENIDOS

Índice de Contenidos i
Relación de Acrónimos iii
Índice de Figuras v

CAPÍTULO 1: INTRODUCCIÓN 1

CAPÍTULO 2: TECNOLOGÍAS Y HERRAMIENTAS EMPLEADAS 13
2.1 INTRODUCCIÓN 13
2.2 JAVA 13
2.3 J2ME 14
2.3.1 ARQUITECTURA J2ME: CONFIGURACIONES Y PERFILES 15
2.3.2 MIDLETS 17
2.3.2.1 MIDP 2.0 17
2.3.2.2 Estructura de ficheros 18
2.3.2.3 Ciclo de vida de un MIDlet 19
2.4 PATRONES DE DISEÑO 20
2.4.1 INTRODUCCIÓN 20
2.4.2 PATRÓN MVC 21
2.4.3 PATRÓN OBSERVADOR 23
2.5 XML 24
2.5.1 XSD 25
2.5.2 PARSERS XML 26
2.6 MODELADO UML 28
2.6.1 INTRODUCCIÓN 28
2.6.2 DIAGRAMA DE CLASES 31
2.6.3 DIAGRAMA DE ESTADOS 31
2.6.4 DIAGRAMA DE SECUENCIA 32
2.7 WI-FI 32
2.8 GPRS 34
2.9 UMTS 36
2.10 ENTORNOS DE DESARROLLO (IDE) 37
2.10.1 ANT 38
2.10.2 OFUSCADOR Y OPTIMIZADOR 39
2.11 EMULADORES 39
2.12 ANALIZADOR DE TRÁFICO 41

CAPÍTULO 3: DISEÑO DE LA APLICACIÓN 43
3.1 INTRODUCCIÓN 43
3.2 VISIÓN GENERAL 43
3.3 FUNCIONAMIENTO DE LA APLICACIÓN 45
3.3.1 PROCESOS DE LA APLICACIÓN 47
3.3.1.1 Conexión al servicio y presentación del juego 47
3.3.1.2 Registro del usuario/partida 49
3.3.1.3 Identificación del usuario/partida 50
3.3.1.4 Ejecución del videojuego 52
3.4 ESTRUCTURA DE LA APLICACIÓN 55
3.4.1 SISTEMA DE ARCHIVOS 55
3.4.1.1 Arquitectura del sistema de archivos 55

3.4.1.2 Archivos de configuración del videojuego 56
3.4.2 CLIENTE 68
3.4.2.1 Introducción 68
3.4.2.2 Sistema de turnos 70
3.4.2.3 Recepción de contenidos en el cliente 73
3.4.2.4 Pérdida de conexión con el servidor 75
3.4.2.5 Vista de la aplicación 75
3.4.2.6 Ofuscación y optimización de la aplicación 77
3.4.3 SERVIDOR 77
3.4.3.1 Introducción 77
3.4.3.2 Control de acceso 79
3.4.3.3 Coherencia del modelo 81
3.4.3.4 Estructura multihilo 85
3.4.3.5 Optimización del tráfico 85
3.4.4 PROTOCOLO IMPLEMENTADO 86
3.4.4.1 Introducción 86
3.4.4.2 Mensajes de flujo de juego 87
3.4.4.3 Mensajes de petición de actualización 95
3.4.4.4 Sockets 97
3.4.4.5 Diagramas de funcionamiento del protocolo 98

CAPÍTULO 4: MANUAL DE USUARIO 107
4.1 MANUAL DE USUARIO ADMINISTRADOR 107
4.1.1 ESPECIFICACIÓN DEL VIDEOJUEGO 107
4.1.2 ESTRUCTURA Y CONTENIDOS 108
4.1.3 PUESTA EN MARCHA DEL SERVIDOR 111
4.2 MANUAL DE USUARIO JUGADOR 113

CAPÍTULO 5: PLAN DE PRUEBAS 123
5.1 MEDICIONES DEL TRÁFICO GENERADO 123
5.1.1 CONSIDERACIONES PREVIAS 123
5.1.2 MEDICIONES REALIZADAS 124
5.2 PLAN DE PRUEBAS DETALLADO 128
5.2.1 PRUEBAS CON EMULADORES 128
5.2.1.1 Pruebas en partida individual 128
5.2.1.2 Pruebas en partida multijugador 131
5.2.2 PRUEBAS CON DISPOSITIVOS MÓVILES REALES 133

CAPÍTULO 6: CONCLUSIONES Y LÍNEAS FUTURAS 135
6.1 CONCLUSIONES FINALES 135
6.2 LÍNEAS FUTURAS 139

BIBLIOGRAFÍA Y REFERENCIAS 141

RELACIÓN DE ACRÓNIMOS

aDeSe	Asociación Española de Distribuidores y Editores de Software de Entretenimiento
AES	Advanced Encryption Standard
AMS	Application Management Software
API	Application Programming Interface
CAAD	Club de Aventuras AD
CDC	Connected Device Configuration
CLDC	Connected Limited Device Configuration
DTD	Document Type Definition
FORTRAN	Formula Translating System
FP	Foundation Profile
GPRS	General Packet Radio Service
GSM	Global System for Mobile communications
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
J2EE	Java 2 Enterprise Edition
J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edition
JAD	Java Application Descriptor
JAR	Java ARchives
JVM	Java Virtual Machine
JSP	Java Server Pages
KVM	Kilobyte Virtual Machine
MIDP	Mobile Information Device Profile
MIMO	Multiple Input – Multiple Output
MMORPG	Massive Multiplayer Online Role-Playing Game
MMS	Multimedia Messaging System
MVC	Modelo Vista Controlador
NPC	Non-Playable Character

OFDM	Orthogonal Frequency Division Multiplexing
OTA	Over The Air
PDA	Personal Data Assistant
PDP-10	Programmed Data Processor model 10
PBP	Personal Basic Profile
PP	Personal Profile
PSP	PlayStation Portable
QoS	Quality of Service
ROM	Read-Only Memory
SDK	Software Development Kit
SGML	Standard Generalized Markup Language
SMS	Short Message Service
SPAC	Sociedad para la Preservación de las Aventuras Conversacionales
SPAG	Society for the Promotion of Adventure Games
SSID	Service Set ID
TCP	Transmission Control Protocol
UML	Unified Modeling Language
UMTS	Universal Mobile Telecommunications System
USB	Universal Serial Bus
W3C	World Wide Web Consortium
WAP	Wireless Application Protocol
WEP	Wired Equivalent Privacy
Wi-Fi	Wireless Fidelity
WLAN	Wireless Local Area Network
WPA	Wi-Fi Protected Access
WPA2	Wi-Fi Protected Access 2
XML	eXtensible Markup Language
XSD	XML Schema Definition

ÍNDICE DE FIGURAS

Figura 1.1 Imágenes de diferentes Aventuras Conversacionales 4
Figura 1.2 Imágenes de la evolución de las Aventuras Conversacionales 7
Figura 2.1 Diferencias entre los compiladores tradicionales y el compilador Java 14
Figura 2.2 Plataforma J2ME, configuraciones y perfiles 16
Figura 2.3 Arquitectura de alto nivel de MIDP 2.0 18
Figura 2.4 Contenido de un fichero de manifiesto *.mf* 19
Figura 2.5 Ciclo de vida de un *MIDlet* 20
Figura 2.6 Relación entre las capas de la arquitectura MVC 22
Figura 2.7 Funcionamiento del patrón observador 24
Figura 2.8 Comparativa de *parsers* pequeños 27
Figura 2.9 Tipos de diagramas UML 30
Figura 2.10 Diagrama de clases de un aeropuerto 31
Figura 2.11 Diagrama de estados de la vida de un trabajador 31
Figura 2.12 Diagrama de secuencia de un proceso de trabajo 32
Figura 2.13 Contenido de un archivo *build.xml* 38
Figura 2.14 Los emuladores Sun WTK y Nokia S60 41
Figura 3.1 Arquitectura de la aplicación 45
Figura 3.2 Diagrama de estados de la aplicación 46
Figura 3.3 Estructura de archivos 56
Figura 3.4 Diagrama de clases del cliente 69
Figura 3.5 Clases que implementan la interfaz de usuario en MIDP 76
Figura 3.6 Diagrama de clases del servidor 78
Figura 3.7 Diagrama de estados del protocolo 89
Figura 3.8 Entrada en partida individual 99
Figura 3.9 Registro en partida individual, entrada o registro en partida multijugador..100
Figura 3.10 Bucle normal de juego 101
Figura 3.11 Usuario que provoca cambios en el modelo o entorno de juego.....102
Figura 3.12 Actualización del contexto de juego 103
Figura 3.13 Abandono de partida 104
Figura 3.14 Fin de partida individual 105
Figura 4.1 Estructura de directorios de cada videojuego.....108

Figura 4.2 Estructura de directorios y llamada al servidor 112
Figura 4.3 Entrada a la aplicación y configuración del servidor..... 114
Figura 4.4 Selección de un juego y presentación de éste 115
Figura 4.5 Entrada en partida individual 115
Figura 4.6 Creación de partida individual 116
Figura 4.7 Entrada en partida multijugador 116
Figura 4.8 Creación de partida multijugador 117
Figura 4.9 Ejemplos de errores en la identificación y registro..... 117
Figura 4.10 Entrada a un juego de la plataforma 118
Figura 4.11 Diversas acciones en el juego 119
Figura 4.12 Diversas acciones con efectos en otros jugadores 120
Figura 4.13 Error en la conexión con el servidor..... 121
Figura 5.1 Ejemplo de tráfico generado por la aplicación 125
Figura 5.2 Medidas de tráfico realizadas 126

CAPÍTULO 1: INTRODUCCIÓN

Hace ya tiempo que los videojuegos dejaron de ser un pasatiempo para jugadores adolescentes, desarrollados por programadores aficionados en pequeños estudios. Desde sus humildes comienzos, allá por 1972 cuando Atari publicó un sencillo juego de tenis de mesa llamado “Pong” [1], habría que esperar hasta casi el final de la década para que esta forma de entretenimiento realmente se popularizara entre el público, lo que ocurrió en 1978 con la salida al mercado de “Space Invaders”. Desde entonces, y con la única excepción de la crisis del sector en 1983 debida al exceso de producción [2], la industria no ha parado de crecer año tras año.

En la actualidad, la aDeSe (Asociación española de Distribuidores y Editores de Software de Entretenimiento) [3], cuyos miembros representan el 80% del negocio de los videojuegos en España, recoge en su informe anual de 2008 que el sector de los videojuegos ya factura 1.432 millones de euros al año, lo que representa el 52% del total de ocio audiovisual e interactivo en nuestro país [4], superando en volumen de beneficios a la suma de las taquillas de cine, las películas en DVD y la música. Según este mismo informe, uno de cada cuatro españoles es consumidor habitual de videojuegos, y en línea con este dato, España es el cuarto país en el que más videojuegos se venden y consumen de Europa, y el sexto del mundo [5].

Para poder mantener el ritmo estable de crecimiento anual, la industria ha tenido que ir adaptándose cada cierto tiempo a las particularidades del mercado, y ha sido explotando nuevas tecnologías o tendencias como ha conseguido prolongar su expansión.

La ubicuidad actual de los terminales móviles ha sido vista por parte de los desarrolladores como una buena oportunidad para explotar sus capacidades cada vez mayores de ejecutar videojuegos. Lejos ya de los antiguos terminales con pantallas monocromo y pocas líneas de texto simultáneas, los móviles actuales tienen pantallas de mayor resolución con miles de colores, así como múltiples posibilidades de conectividad (Wi-Fi, Bluetooth, etc.) que permiten que el jugar ya no tenga que ser una experiencia solitaria. De hecho, el acceso a Internet por Wi-Fi desde un teléfono móvil aumentó un 132%, con el mayor crecimiento dándose en España e Italia [6].

Además, en la actualidad no es necesario acudir a las gamas más altas (y por tanto más costosas) de los fabricantes para encontrarnos con terminales de esas características. El tiempo medio de renovación de un móvil por parte de un usuario está en torno a los 2 años, y además el 42% no gasta dinero en adquirir su nuevo terminal [7].

Con todo lo anterior, tenemos una población que mantiene actualizado constantemente su teléfono a los últimos modelos, y que incluso los dispositivos que se adquieren sin desembolso de dinero para el usuario, ya vienen equipados con muchas de las últimas novedades y mejoras en conectividad.

Se estima que los españoles comprarán 17 millones de videojuegos para móviles durante 2009, siendo esta cifra un 20% superior a la del año anterior [8]. Visto esto, no es de extrañar que el desarrollo de videojuegos para este tipo de dispositivos se haya convertido en una prioridad para las compañías.

La variedad de títulos disponibles para teléfonos móviles actualmente es muy elevada, viéndose beneficiada de la multitud de consolas portátiles existentes (PSP, Nintendo DS, N-Gage) y la similitud de desarrollar para ese tipo de plataformas con limitaciones. Podemos encontrar títulos de muy diversos géneros, desde juegos arcade, de estrategia o simulación, pasando por juegos de aventuras. Centrándonos en este último caso, los inicios de los juegos de aventuras vinieron de la mano de las llamadas aventuras conversacionales.

Las aventuras conversacionales fueron los primeros juegos llevados al ordenador, en una época en que las capacidades gráficas de los ordenadores eran inexistentes o muy limitadas. De aquí que, en su origen y forma más purista, las aventuras conversacionales no necesitaran mostrar ningún gráfico.

En su forma más extendida, una aventura conversacional presenta al jugador una historia dentro de la cual se van a desarrollar todas las acciones del juego. La dinámica de juego consiste en ir mostrando localizaciones al usuario, que estarán formadas por un texto descriptivo, acompañado de una imagen que lo enriquezca y aumente la sensación de inmersión. Dichas localizaciones estarán conectadas entre sí formando un gran mapa, ya sea de forma directa o por medio de artefactos que posibiliten esa unión como pueden ser escaleras o puertas.

En cada localización, al usuario se le presentan una serie de acciones a realizar. Estas acciones pueden ser escritas por el jugador en lenguaje natural, que sería interpretado y procesado por un *parser* o analizador sintáctico, que se encargaría de analizar y convertir la acción escrita en una de las acciones realizables en el juego, o en su caso, de informar de su imposibilidad al no estar contemplada en la programación del juego. En el caso de dispositivos en los que no está disponible de forma directa un periférico específico para introducir texto, como por ejemplo consolas y móviles, las acciones se pueden presentar a modo de lista navegable por medio de un cursor. De una u otra forma, una vez elegida la acción, el juego se encarga de procesarla y de producir una serie de consecuencias, que modificarían la localización actual, o nos llevarían a otra situación. Todas estas situaciones son coherentes con el argumento del juego y tienen una relación entre sí, cuanto mejor sea ésta, mayor será la calidad de la aventura y, en definitiva, el disfrute del jugador. Aunque no sólo será necesaria la existencia de una estructura básica de situaciones y acciones que provoquen modificaciones que hagan avanzar al argumento, sino también la riqueza de acciones que divergen de la estructura lineal del juego, creando una experiencia más elaborada y presentando un desafío al jugador, que ha de ir probando para encontrar el camino que hace que progrese la trama.

Además de las localizaciones y las acciones, otro punto importante de las aventuras conversacionales está en los objetos. Éstos se encuentran repartidos por distintos lugares del mapa, y pueden ser examinados, cogidos, dejados y manipulados solos o junto a otros objetos por parte del jugador, que habrá de utilizar su ingenio y dotes deductivas para usarlos para resolver las situaciones que se le plantean. Los objetos van desde puertas que dan paso a otras localizaciones, a artilugios que son imprescindibles para progresar (por ejemplo, una linterna que, usada en el lugar oportuno, nos permite atravesar un pasadizo oscuro), pasando por NPC (Non-Playable Character), que son personajes introducidos en la historia y que son manejados por el propio juego, con los que el usuario debe interaccionar para poder seguir el curso de la aventura.

Podemos ver algunos ejemplos representativos de aventuras conversacionales en la Figura 1.1, donde se muestra la interfaz principal de juego, compuesta por una localización junto con la descripción que la acompaña, en diversos títulos.

Yenght Carvalho Jabato

Figura 1.1 Imágenes de diferentes Aventuras Conversacionales

En la figura se puede apreciar el refinamiento progresivo en el interfaz de este tipo de juegos desde comienzo de los 80 hasta finales de esa década, en la que tuvo lugar un mayor desarrollo de este género. En los dos primeros casos, Yength (la primera aventura conversacional que llegó al mercado español) y Carvalho, nos encontramos con un formato de presentación más sencillo, consistiendo únicamente en una imagen que ilustra la localización actual, así como el texto descriptivo que la acompaña. En el tercer ejemplo, Jabato, los gráficos son de mejor calidad, y la interfaz es más sofisticada: podemos apreciar las caras de dos personajes o NPC a la derecha de la pantalla, que el jugador podía controlar con sus acciones, y cuya ayuda combinada con sus especiales características eran imprescindibles para acabar la aventura con éxito.

El origen de las aventuras conversacionales tuvo lugar en 1975, cuando Will Crowther, con la ayuda del estudiante Don Woods, escribió “Adventure” usando el lenguaje de programación FORTRAN. El juego original no disponía de ningún gráfico, y estaba diseñado para ejecutarse en un mainframe PDP-10. Cuando comenzó a popularizarse, “Adventure” fue portado a otros sistemas, y de hecho continúa siendo una de las aventuras conversacionales con más conversiones y expansiones hechas, incluida la española “La Aventura Original” (1988), de Aventuras AD. A finales de los años 70 y comienzos de los 80, el género se popularizó en los países de habla inglesa, con una gran cantidad de lanzamientos llegando al mercado, arropados por la buena acogida.

En España, habría que esperar hasta que comenzaron a surgir desarrollos en nuestro idioma, siendo el primer título en ser publicado en nuestro país el anteriormente nombrado “Yenght”, de la recién fundada primera empresa de videojuegos española, Dinamic.

Más adelante, las aventuras conversacionales se harían realmente populares en nuestro país a partir de 1988, cuando Dinamic decidió dedicar una división exclusivamente al desarrollo de este tipo de juegos, Aventuras AD.

Aventuras AD fue la única empresa dedicada en exclusiva a las aventuras conversacionales, y una de las más representativas de la llamada "edad de oro del software español". Durante su periplo entre 1988 y 1992, publicó siete juegos a la venta, ante un gran éxito de público y crítica, entre los cuales se hayan "La Aventura Original", adaptación libre de "Adventure", "Jabato" (1989), que incluía la novedad del manejo de múltiples personajes, "La Aventura Espacial" (1990), y la trilogía de Ci-u-Than, formada por "Cozumel" (1990), "Los Templos Sagrados" (1991) y "Chichén Itzá" (1992), siendo éste último considerado por muchos el juego más pulido y completo, que contaba con una gran cantidad de NPC y puzzles en su amplio mapa. Sin embargo, fue lanzado al mercado en medio de la crisis de los ordenadores de 8 bits, y justo cuando acababan de irrumpir en escena las aventuras gráficas, herederas del trono de las aventuras conversacionales, y que sustituían el añejo sistema de escritura de acciones por uno que hacía uso de la percepción natural humana, el "señalar y hacer clic".

Aún así, hoy en día, gracias a los repositorios en Internet, se pueden encontrar la mayoría de los títulos de aventuras conversacionales disponibles para su uso con distintos emuladores, que simulan el comportamiento de los antiguos ordenadores de 8 y 16 bits para los que fueron programadas. Además de esto, y gracias a las facilidades que proporciona Internet, existen clubes dedicados a las aventuras conversacionales como es el caso de CAAD (Club de Aventuras AD) [9], donde se posibilita la reunión de aficionados al género y el acceso a material relacionado, artículos, listas de correo, imágenes ROM de aventuras para emuladores, etc. También destacan los concursos de mini-aventuras que se realizan, diseñadas por aficionados al género en su tiempo libre [10]. Para estar al tanto de las últimas noticias y la actualidad del mundillo, se encuentra el fanzine de la SPAC (Sociedad para la Preservación de las Aventuras Conversacionales) [11]. Fuera de nuestras fronteras, tenemos a su equivalente en el fanzine inglés SPAG (Society for the Promotion of Adventure Games) [12], que mantiene una viva comunidad de lectores y editores, que contribuyen a que el género no caiga en el olvido.

Las aventuras conversacionales evolucionaron a otro tipo de juegos por dos caminos: por una parte y como se ha comentado, están las aventuras gráficas, en las que el interfaz y

la forma de interaccionar del jugador pasa a ser más intuitiva, ya que en lugar de limitarse a dar órdenes introducidas por teclado o seleccionando de una lista, se usan unos gráficos más detallados y un cursor que se muestra sobre la pantalla, y que el jugador es libre de mover para señalar e interactuar con el entorno. Un exitoso ejemplo de este tipo de juegos es "Myst" (1993), que se mantuvo como el título más vendido para PC de la historia durante una década [13].

La otra dirección de evolución de las aventuras conversacionales tiene que ver con los jugadores, más que con la interacción. Siendo juegos diseñados para un único jugador, la idea era inspirarse en los juegos de rol de lápiz y papel, y hacer una traslación al mundo de los ordenadores de este sistema multijugador. Así, existe un mundo ficticio dentro del juego en el que los jugadores son representados por medio de avatares o personajes, y que tiene entidad propia independientemente de que se esté jugando o no. De ahí viene el concepto de "mundo persistente", un escenario de juego que sigue existiendo aunque un usuario se desconecte, y en el que seguirán ocurriendo acciones realizadas por el resto de jugadores, cuyas consecuencias serán visibles cuando se retorne al juego. En esencia, es un mundo en el que siguen ocurriendo cosas mientras que el jugador no está jugando, lo cual lo dota de más realismo y permite la interacción con otros jugadores.

Este tipo de juegos, (también llamados MMORPG, del inglés *Massive Multiplayer Online Role-Playing Game*, juegos de rol multijugador masivo online) alcanzaron la popularidad a mediados de los 90 con títulos como "Ultima Online" y "Neverwinter Nights", y continúan siendo muy populares en la actualidad, teniendo su máximo exponente en "World of Warcraft", que tiene 11.5 millones de subscriptores de pago mensual, y que por sí solo acapara el 62% del mercado de este género [14].

En la Figura 1.2 podemos ver un ejemplo de estas dos direcciones de evolución de las aventuras conversacionales, con la aventura gráfica "The Secret of Monkey Island" (1990) de LucasArts, en la que vemos la lista de acciones a realizar abajo, mientras un cursor señala el lugar u objeto sobre el que aplicar la acción escogida en el escenario. Y por otra parte, tenemos a "Neverwinter Nights", juego multijugador masivo que fue distribuido por America Online y mantenido como un mundo persistente desde 1991 hasta 1997.

The Secret of Monkey Island

Neverwinter Nights

Figura 1.2 Imágenes de la evolución de las Aventuras Conversacionales

Teniendo en cuenta las cifras tan favorables que presenta el mercado de los videojuegos actualmente y con unas previsiones futuras que no son menos halagüeñas, la ubicuidad de los dispositivos móviles con amplias posibilidades de conectividad y la aceptación de su uso como terminales de ocio electrónico, así como las especiales características de los juegos de aventura conversacional que hacen que su adaptación a las limitaciones de este tipo de terminales sea excelente; en el presente proyecto se pretende adaptar un motor de videojuegos conversacionales sobre WAP para dispositivos móviles [15], para que la interacción se lleve a cabo desde una aplicación J2ME en el terminal móvil, que se conecte de forma inalámbrica utilizando Wi-Fi, por medio de un punto de acceso, u otros medios como GPRS o UMTS; a un servidor de Internet que sirva los contenidos, posibilitando de forma sencilla la incorporación de varios jugadores simultáneos a la misma partida del juego, así como una comunicación bidireccional que permita mantener un mundo persistente en el que las acciones que un jugador realiza, tengan influencia sobre el resto de jugadores.

El motor original se desarrolló siguiendo la arquitectura MVC, que separa de forma clara los contenidos de la aplicación en las tres partes que le dan nombre: Modelo, Vista y Control. El modelo estaba formado por un sistema de archivos XML (eXtensible Markup Language) que era el encargado de configurar desde los textos e imágenes a mostrar, hasta los objetos y acciones a realizar para cada uno de los videojuegos disponibles y de personalizar éstos para cada uno de los usuarios activos. Por su parte la vista, estaba compuesta por una serie de archivos JSP (Java Server Pages) que adaptaban los contenidos a mostrar a las características específicas de cada usuario. Por último, el control estaba formado por un conjunto de *servlets* Java que implementaban la inteligencia de la aplicación.

La comunicación en el caso del motor original era de tipo petición-respuesta, de forma unidireccional desde el terminal móvil hacia el servidor usando una conexión sobre WAP y el protocolo HTTP. De modo que el servidor enviaba una página web HTML, que se había generado en él de forma dinámica, al dispositivo móvil, y era el navegador web de éste el encargado de automáticamente interpretarla y mostrarla en la pantalla del terminal, para presentar al usuario la información sobre el escenario y una lista de acciones a realizar. Se utilizaba scripting del lado del cliente para realizar una serie de tareas (como la generación de dicha lista de acciones) y otras comprobaciones (por ejemplo, para ver si algún campo quedaba vacío y avisar de ello). Una vez hecha una elección sobre la acción a tomar, se enviaba una petición en la que se indicaban los cambios a realizar en el modelo, de lo que se encargaban los *servlets* que formaban el control en el servidor. A continuación estos *servlets* calculaban las consecuencias a estos cambios y reenviaban la situación actualizada de nuevo a la aplicación móvil. Esta forma de proceder limitaba la posibilidad de múltiples jugadores compartiendo una misma partida de forma simultánea, ya que el proceso era unidireccional y no permitía la propagación de los cambios realizados por un usuario por parte del servidor al resto de jugadores.

En el presente proyecto se pretende adquirir las competencias necesarias en las tecnologías utilizadas en el desarrollo del motor original, entre ellas los *servlets* o servidores de aplicaciones Java, la creación dinámica de páginas webs con JSP, o la generación y tratamiento de documentos XML. Todo ello con el fin de lograr una profunda compresión de la aplicación original, que llegado el momento nos permita realizar los cambios necesarios para adaptarla al propósito del presente proyecto.

Para llevar la adaptación a cabo, y aprovechando la división lógica que propicia el uso de la arquitectura MVC en el motor original, será necesario eliminar por completo la vista realizada por medio de JSP, y crear un *MIDlet* J2ME que presente al usuario una interfaz de juego con las características propias de un videojuego conversacional, que se pueda adaptar a distintos títulos por medio de la modificación de los archivos de configuración XML del juego. El uso de J2ME permite tener una tecnología soportada por la mayoría de dispositivos móviles del mercado, y la facilidad de que es multiplataforma, por lo que no es necesario más que desarrollar una única versión para todos los dispositivos compatibles. Además, se hará uso de la comodidad que supone para el desarrollo el hecho de poder usar emuladores para ir viendo en todo momento cómo se comporta la aplicación, sin necesidad

de tener que hacer pruebas en terminales móviles hasta la fase final, con el ahorro de tiempo y hardware que ello supone.

A su vez, será necesaria una profunda adaptación de la parte de control en el servidor realizada con *servlets*, para convertirla en un servidor Java que, por una parte mantenga la generación de los archivos XML correspondientes al modelo del motor original, pero a su vez haga de interfaz con la nueva aplicación desarrollada para el terminal móvil, que hará uso de una comunicación bidireccional que nos permita fácilmente la incorporación de varios jugadores simultáneos a un mismo mundo, sobre el que se pueden realizar acciones que tengan consecuencias que el resto de jugadores puedan apreciar. Así, habrá que eliminar la conexión unidireccional del protocolo HTTP, y hacer que ésta se lleve a cabo de forma bidireccional.

Este servidor Java, que pasaría a llevar la parte de control de la aplicación, habrá que diseñarlo teniendo en cuenta unos requisitos de concurrencia que impone el diseño de cualquier sistema multijugador, de forma que se proteja el acceso al modelo formado por los archivos XML para que sólo se puedan realizar accesos ordenados y autorizados, y para que haya una propagación de cambios que garantice que todos los jugadores tienen en todo momento una versión actualizada del escenario de juego.

La conexión entre la aplicación móvil y el servidor se adaptará para aprovechar las posibilidades de conectividad cada vez mayores que nos proporcionan los terminales móviles, usando Wi-Fi, GPRS o UMTS en lugar de WAP, ya que mantener una conexión sobre WAP añade unos costes por tráfico que no son nada desdeñables, y en la actualidad es común encontrar puntos de acceso inalámbricos abiertos dando servicio a la población tanto en lugares públicos como aeropuertos o estaciones, como en lugares de ocio tales como cafeterías, restaurantes o centros comerciales. También están bastante extendidas las tarifas planas para conexiones de datos como GPRS o UMTS, que posibilitan el uso de Internet desde el móvil a un coste mensual razonable y fijo, si no se sobrepasan unos límites de tráfico establecidos por las operadoras.

Por su parte, se ampliará la funcionalidad del motor original con nuevas acciones como hablar con un NPC o dar un objeto a un NPC, por lo que será necesario modificar el modelo formado por los archivos XML. De este modo, en los juegos escritos para el motor de videojuegos conversacionales propuesto en este proyecto, se irá un paso más allá y se

incluirá la posibilidad de tener NPC con una cierta inteligencia, que puedan conversar con el jugador siguiendo un guión preestablecido que hará que la conversación se dirija hacia una u otra conclusión dependiendo de lo que se diga en ella. De esta forma, si la conversación toma un tono cordial, el NPC puede decidir entregarle algún objeto al jugador que le ayude en la historia; mientras que si el jugador decide adoptar una actitud hostil en el diálogo, se puede encontrar con consecuencias negativas para su personaje, llegado el caso incluso la muerte. Este guión de los NPC se incluirá dentro de los archivos de configuración XML del juego, usando para ello una serie de nuevas etiquetas y estructuras creadas para implementar esta funcionalidad.

Asimismo, cabe resaltar que estas nuevas acciones añadidas al motor original también tienen sentido en un contexto multijugador (por ejemplo, dar un objeto o hablar con otro usuario con el que se comparte escenario). Es por ello que también se ha añadido la lógica necesaria al motor para poder tener una interacción entre usuarios dentro de una misma partida, que junto a la inclusión de NPC inteligentes, hacen que el mundo creado por estos juegos se vea más vivo y la historia pueda tener más profundidad.

Por último, en cuanto a la comunicación entre el servidor y los terminales móviles, se sustituirá el binomio WAP/HTML utilizado por el motor original por un protocolo propio desarrollado para este proyecto. Este protocolo estará basado en una comunicación por sockets a través de los que se enviarán mensajes de control de flujo (que regularán el funcionamiento de una máquina de estados), y datos (tanto texto como imágenes), por lo que no se hará uso de ninguna facilidad adicional (como un navegador web, por ejemplo) proporcionada por los terminales para ayudar a la presentación de esta información.

El presente documento está estructurado en diversos capítulos para poder explicar con detalle todas las partes del proyecto:

1. Introducción: es el presente capítulo, en el que se hace una somera descripción y justificación del proyecto, así como de los objetivos a conseguir.
2. Tecnologías y herramientas empleadas: en el cual se detallan las herramientas software y hardware usadas, comparándolas con otras similares y justificando su elección en base a sus ventajas e inconvenientes.
3. Diseño de la aplicación: donde se explica en detalle la adaptación realizada al motor original, hasta llegar a la aplicación final, de la que se detallan las

diferentes partes que la componen, y se realiza un estudio pormenorizado de su funcionamiento.

4. Instalación y manual de usuario: en el que se explica paso a paso cómo instalar cada uno de los elementos que conforman el sistema para su correcto funcionamiento.
5. Plan de pruebas: donde se detallan cada una de las pruebas realizadas al sistema para comprobar su correcto funcionamiento.
6. Conclusiones y líneas futuras: en el que se hace un balance del proyecto, así como el estudio de posibles mejoras o ampliaciones.

CAPÍTULO 2: TECNOLOGÍAS Y HERRAMIENTAS EMPLEADAS

2.1 INTRODUCCIÓN

Para la realización de este proyecto, ha sido necesario el empleo de una serie de herramientas tanto software (en su mayoría) como hardware. A lo largo de este capítulo se explicará en detalle el motivo de su elección frente a otras alternativas, así como la labor para la que fueron utilizadas en el ámbito del proyecto que nos ocupa.

Cabe reseñar que todas las herramientas que se emplearon en el presente proyecto son de libre uso y distribución, siguiendo la filosofía general de búsqueda de la estandarización y el mínimo coste global, tanto para el desarrollador como para el cliente final.

2.2 JAVA

Java tuvo unos comienzos dubitativos, hasta que en 1994, en Sun Microsystems reconocieron que la mejor forma de enfocar su desarrollo futuro era orientándolo hacia la Web. Desde entonces y hasta estos días, debido a unas características que se discutirán a continuación, Java floreció como uno de los lenguajes con más implantación, gozando actualmente de una excelente salud y de infinidad de desarrolladores, herramientas y una base de código extensísima.

Las características que hacen de Java un lenguaje tan popular son:

- Es independiente de la plataforma. Es decir, una aplicación una vez escrita, puede ser ejecutada en cualquier dispositivo, tal como reza el axioma de Java: "*write once, run anywhere*". Esto es así porque cuando se compila una aplicación Java, se genera un código intermedio llamado *bytecode*, que ha de ser interpretado más tarde por la Máquina Virtual Java (JVM – *Java Virtual Machine*), ésta sí dependiente del sistema operativo, presente en la gran mayoría de sistemas. En la Figura 2.1 se muestra la diferencia entre la compilación tradicional dependiente de la máquina y la generación de los *bytecodes* multiplataforma.

- Seguridad. Esto es en parte debido a la generación de los *bytecodes*, ya que en Java, por su propio diseño, no se permite acceder a la máquina físicamente más que a través de librerías que sí dependen del sistema operativo y que no son estándares de Java.
- Es software libre, y los IDE (Entornos de Desarrollo Integrados, del inglés *Integrated Development Environment*) más conocidos para programar en Java son gratuitos, como NetBeans y Eclipse. Por lo que además de las ventajas que ofrece su desarrollo, también se pueden realizar desarrollos en Java con costes asociados muy pequeños.

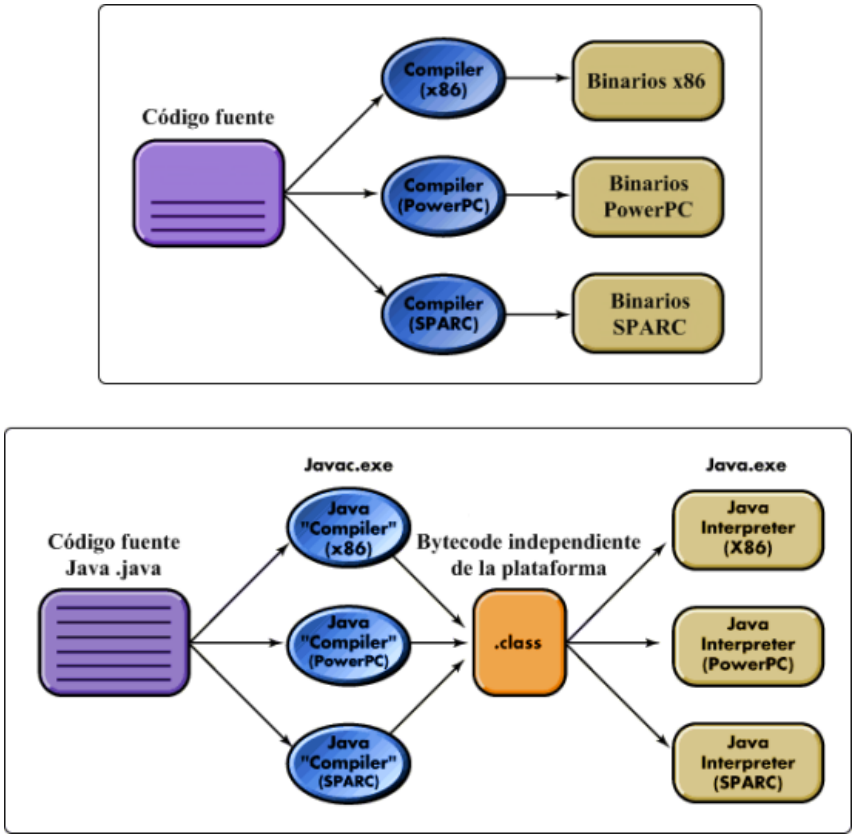


Figura 2.1 Diferencias entre los compiladores tradicionales y el compilador Java

2.3 J2ME

J2ME (Java 2 Micro Edition) fue diseñado por Sun Microsystems como un subconjunto de las especificaciones de Java, que definen una versión minimizada de la plataforma Java 2 Standard Edition (J2SE), que se ajusta especialmente bien a dispositivos con pocos recursos o con capacidades restringidas, como pueden ser teléfonos móviles, PDA (agendas electrónicas), o sistemas embebidos.

2.3.1 ARQUITECTURA J2ME: CONFIGURACIONES Y PERFILES

Como se puede ver, J2ME está pensado para cubrir un gran número de dispositivos con características muy diversas, y, por lo tanto, demanda una arquitectura capaz de gestionar de forma adecuada las funcionalidades disponibles en cada familia de dispositivos que lo soportan. Debido a la diferencia de restricciones y capacidades en todos los dispositivos soportados, J2ME se estructura en configuraciones, perfiles, y ciertos paquetes opcionales. Además de estos elementos, J2ME también incluye el propio lenguaje de programación, y una máquina virtual.

En la Figura 2.2 se puede ver cómo se estructura en la actualidad la plataforma J2ME. En la base del diagrama, se sitúan dos configuraciones básicas, que trabajan sobre una máquina virtual:

- CLDC (*Connected Limited Device Configuration*) es la configuración estándar para los dispositivos inalámbricos con capacidades más limitadas, y también es la más extendida. Esta configuración proporciona un nivel mínimo de funcionalidades para desarrollar aplicaciones para un determinado conjunto de dispositivos como los teléfonos móviles o PDA de capacidades limitadas. Por tanto, CLDC provee de un conjunto de clases esenciales para construir aplicaciones para este tipo de dispositivos. En este caso, se trabaja sobre una máquina virtual llamada KVM (*Kilobyte Virtual Machine*), que es más pequeña y por lo tanto menos potente que la dedicada a entornos mayores (JVM).

- CDC (*Connected Device Configuration*) es otra configuración pensada para dispositivos con una capacidad de proceso relativamente mayor (por ejemplo, un mínimo de 2 megabytes de memoria, así como capacidades de red más potentes que las correspondientes a CLDC). Además, en CDC y su perfil correspondiente, la máquina virtual que se utiliza es la misma que la utilizada en entornos superiores (JVM), como el ya nombrado J2SE, y J2EE (Java 2 Enterprise Edition).

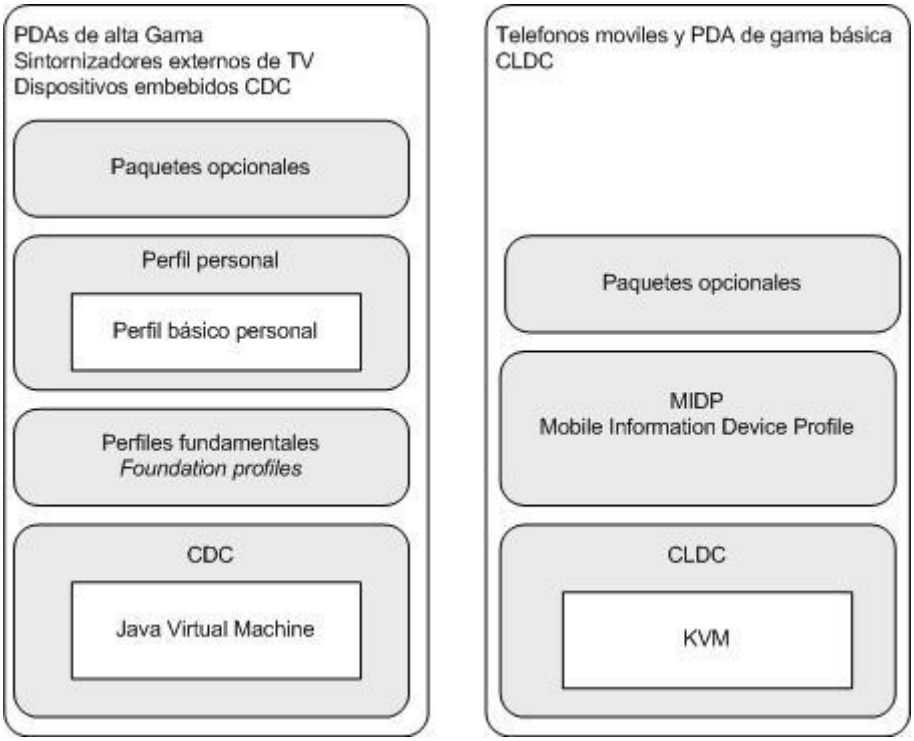


Figura 2.2 Plataforma J2ME, configuraciones y perfiles

Por encima de las configuraciones, están los perfiles. El perfil es un grupo más específico de API (del inglés *Application Programming Interface*, Interfaz de Programación de Aplicaciones) desde el punto de vista del dispositivo. Es decir, la configuración da cobertura a una familia de dispositivos más o menos amplia, y el perfil se orienta a un determinado tipo de dispositivos dentro de esa familia. El perfil añade una serie de funcionalidades además de las que ya aporta la configuración, por lo que al final, entre unas aportaciones y otras, se tiene un conjunto de funcionalidades adecuadas al grupo específico de dispositivos en cuestión. Cabe reseñar, llegados a este punto, que las aplicaciones desarrolladas sobre un determinado perfil van a ser portables a cualquier dispositivo que soporte ese perfil. Sobre una configuración pueden existir diversos perfiles.

Sobre CDC están los perfiles FP (*Foundation Profiles*, Perfiles Fundacionales), que son los perfiles de más bajo nivel, y están pensados para elementos con capacidades de red pero sin interfaz de usuario, por lo que son especialmente adecuados para sistemas embebidos. Estos perfiles se pueden combinar con el que tienen justo por encima en el diagrama, el perfil PP (*Personal Profile*, Perfil Personal), para dispositivos que requieran de una interfaz gráfica de usuario. Además, hay disponible también una versión reducida del perfil PP, que se denomina PBP (*Personal Basic Profile*, Perfil Básico Personal) y está

diseñado para un entorno de trabajo que necesite un nivel básico de presentación de información gráfica.

Por otra parte, sobre CLDC se presenta la especificación MIDP (*Mobile Information Device Profile*, Perfil para Dispositivos de Información Móviles), que está especialmente pensada para trabajar sobre teléfonos móviles y PDA. Dentro de la especificación, se define un dispositivo MIDP como un dispositivo pequeño de recursos limitados, móvil y con una conexión inalámbrica. Es decir, las funcionalidades que provee hacen referencia a conexiones de red limitadas en velocidad y estabilidad, interfaz de usuario reducida y almacenamiento local de información en el dispositivo.

2.3.2 MIDLETS

Será sobre este último, el perfil MIDP, y su configuración subyacente (CLDC), sobre los que se trabaje en el presente proyecto. El motivo es que es el perfil que está disponible en teléfonos móviles y PDA, y a día de hoy es el perfil más extendido e implantado. La última versión disponible de la especificación MIDP es la 2.1, aunque buscando una mayor compatibilidad en la aplicación desarrollada, y teniendo en cuenta el hecho de que la versión 2.1 no aporta diferencias sustanciales con respecto a la versión anterior desde el ámbito del desarrollo de este proyecto, se utilizará la versión MIDP 2.0. En esta versión se incluyen una serie de elementos exclusivos que hacen más fácil el desarrollo de videojuegos, que no estaban en las versiones anteriores.

2.3.2.1 MIDP 2.0

Los requisitos mínimos que ha de cumplir un dispositivo para poder ejecutar aplicaciones MIDP 2.0 son:

- Pantalla de, al menos, 96x54 píxeles, 1 bit (blanco y negro), proporción del aspecto del píxel (aproximadamente 1:1).
- Entrada por teclado o pantalla táctil.
- 256 KB de memoria no volátil para la aplicación MIDP (sin tener en cuenta la requerida por CLDC), 8 KB de memoria no volátil para los datos persistentes creados en la aplicación, y 128 KB de memoria volátil para el entorno de ejecución de la máquina virtual.
- Conexión bidireccional, acceso inalámbrico posiblemente no permanente, con ancho de banda limitado.

- En cuanto a sonido, capacidad para reproducir tonos, ya sea por hardware o software.

En la Figura 2.3 se muestra la arquitectura a alto nivel que provee MIDP 2.0:

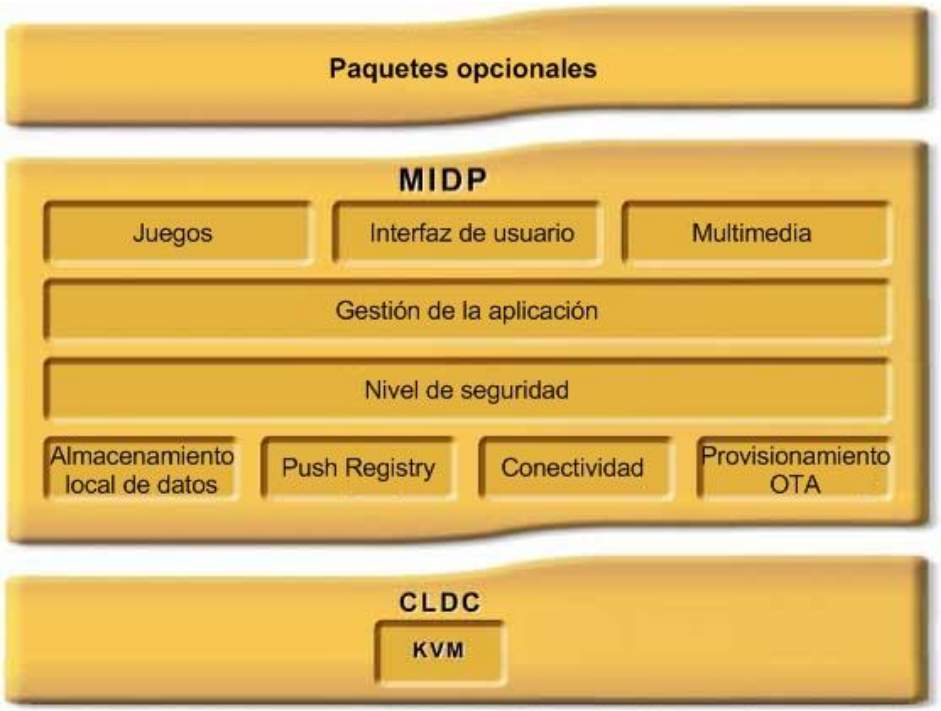


Figura 2.3 Arquitectura de alto nivel de MIDP 2.0

Las aplicaciones J2ME que han sido desarrolladas bajo la especificación MIDP, se denominan *MIDlets*. Las clases de un *MIDlet* se almacenan en el formato definido por los *bytecodes* Java, dentro de un fichero *.class*. Estas clases han de pasar por un proceso de preverificación en tiempo de desarrollo que garantice que no realizarán ninguna operación no permitida cuando estén funcionando en el terminal móvil. Este proceso no se puede hacer en tiempo de ejecución por las limitaciones inherentes a la máquina virtual que se utiliza (KVM) que, liberada de esta tarea, pasa a ser más pequeña y eficiente. La preverificación se realiza después de la compilación, y el resultado es la nueva clase.

2.3.2.2 Estructura de ficheros

Los *MIDlets* están pensados para poder ser descargados a través de Internet. Este tipo de descarga recibe el nombre de OTA (*Over The Air*), y está compuesta por un archivo con extensión *.jar* (*Java Archive*, Archivo Java) en el que se empaqueta la aplicación en sí (clases del *MIDlet*, clases de apoyo, recursos como imágenes o sonidos), así como un

fichero denominado manifiesto; y un fichero descriptor con extensión *.jad* (*Java Application Descriptor*, Descriptor de la Aplicación Java).

En el fichero de manifiesto (de extensión *.mf*), se describe el contenido del fichero *.jar*. En la Figura 2.4 se muestra el contenido de un manifiesto típico:

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.1
Created-By: 11.0-b15 (Sun Microsystems Inc.)
MIDlet-1: JuegoCliente,/icon.png,JuegoCliente
MIDlet-Permissions: javax.microedition.io.Connector.socket
MIDlet-Vendor: Marcos Toledo Pariente
MIDlet-Name: JuegoCliente
MIDlet-Version: 1.3
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-2.0
```

Figura 2.4 Contenido de un fichero de manifiesto *.mf*

El programa encargado de manejar la descarga de *MIDlets* y de controlar los estados por los que éstos pasan mientras se están ejecutando en el dispositivo se denomina AMS (*Application Management Software*, Software de Dirección de Aplicaciones). Estos estados son los siguientes: Localización, Instalación, Ejecución, Actualización y Borrado.

El fichero descriptor *.jad* proporciona la información requerida por el AMS para llevar a cabo la descarga y asegurarse de que ese *MIDlet* es adecuado para el dispositivo en el que se quiere instalar. Es importante destacar que hay que tener cuidado con la procedencia de los ficheros *.jar* que se instalan en los dispositivos, ya que si ésta no es fiable, podrían contener código malicioso no verificado que escapase al control de la máquina virtual, pudiendo causar perjuicios al usuario (por ejemplo, mediante el envío de mensajes SMS no autorizados sin que el usuario se percate).

2.3.2.3 Ciclo de vida de un *MIDlet*

Durante su ciclo de vida activo, un *MIDlet* pasa por varios estados, desde que se crea hasta que se destruye, devolviendo todos los recursos que le había proporcionado el sistema. En la Figura 2.5 se muestra un diagrama con estos estados y los métodos que regulan la transición de un estado a otro.

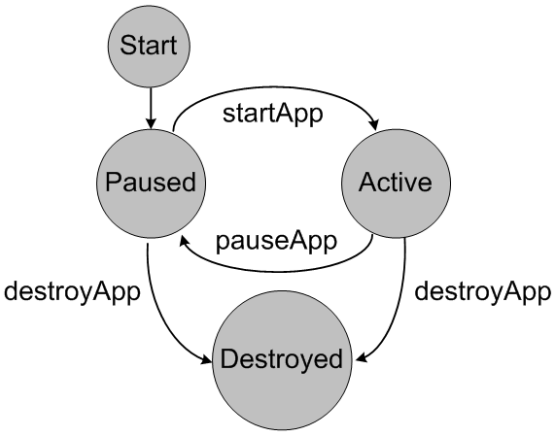


Figura 2.5 Ciclo de vida de un *MIDlet*

Los estados posibles en los que se puede encontrar un *MIDlet* son:

- Detenido (*Paused*), estado en el que se encuentra un *MIDlet* que ha sido creado, pero aún no ha sido ejecutado por primera vez el método *startApp()*. También se puede provocar este estado a través de una llamada al método *pauseApp()*. En este estado, el *MIDlet* mantiene los mínimos recursos posibles y admite cualquier modificación asíncrona.
- Activo (*Active*), estado de ejecución del *MIDlet* en el que se entra tras la llamada inicial a *startApp()*.
- Destruido (*Destroyed*), estado provocado por la invocación del método *destroyApp()*. Una vez que el *MIDlet* entra en este estado, no podrá hacer una transición a ningún otro estado, habrá terminado toda su actividad.

2.4 PATRONES DE DISEÑO

2.4.1 INTRODUCCIÓN

Los patrones de diseño (*design patterns*) son la base para la búsqueda de soluciones a problemas comunes en el desarrollo del software y otros ámbitos referentes al diseño de interacción o interfaces.

Un patrón es una solución a un problema de diseño. Para que una solución pueda ser considerada un patrón, debe reunir una serie de características. Principalmente se debe haber comprobado su efectividad resolviendo problemas similares en situaciones anteriores y debe ser reusable, de modo que se pueda aplicar a diferentes problemas de diseño en distintas circunstancias.

Estos patrones tienen su origen en el mundo de la arquitectura, cuando en 1977 Christopher Alexander los propuso como forma de lograr una construcción de edificios de mayor calidad en base a seguir una serie de directrices que solucionan un problema que ocurre frecuentemente [16]. Así, sería a principios de los 90 cuando los patrones de diseño cosecharían un gran éxito en el campo de la informática, tras la publicación del libro "Design Patterns" [17], en el que se recogían 23 patrones de diseño comunes.

Los objetivos principales de los patrones son:

- Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Formalizar un vocabulario común entre diseñadores.
- Estandarizar el modo en el que se realiza el diseño.
- Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimientos ya existentes.

No es obligatorio usar los patrones, solo es aconsejable en el caso de que se tenga el mismo problema o uno similar al que soluciona el patrón.

2.4.2 PATRÓN MVC

MVC (Modelo Vista Controlador) es un patrón de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos. El patrón MVC se ve frecuentemente en aplicaciones web, donde la vista es la página HTML y el código que provee de datos dinámicos a la página. El modelo es el sistema de gestión de base de datos, y el controlador es el responsable de recibir los eventos de entrada desde la vista.

Modelo: Es la representación específica de la información con la cual el sistema opera, por ello, define la funcionalidad del sistema. Si estamos ante un modelo activo, notificará a las vistas los cambios que pueda producir en los datos un agente externo.

Vista: Presenta el modelo en un formato adecuado para interactuar, usualmente la interfaz de usuario.

Controlador: Responde a eventos, usualmente acciones del usuario, e invoca cambios en el modelo y probablemente en la vista. Por ello, se considera la inteligencia de un sistema que implemente la arquitectura MVC.

Aunque existen distintas implementaciones de MVC, se puede ver en la Figura 2.6 el diagrama de estados que sigue la más común:

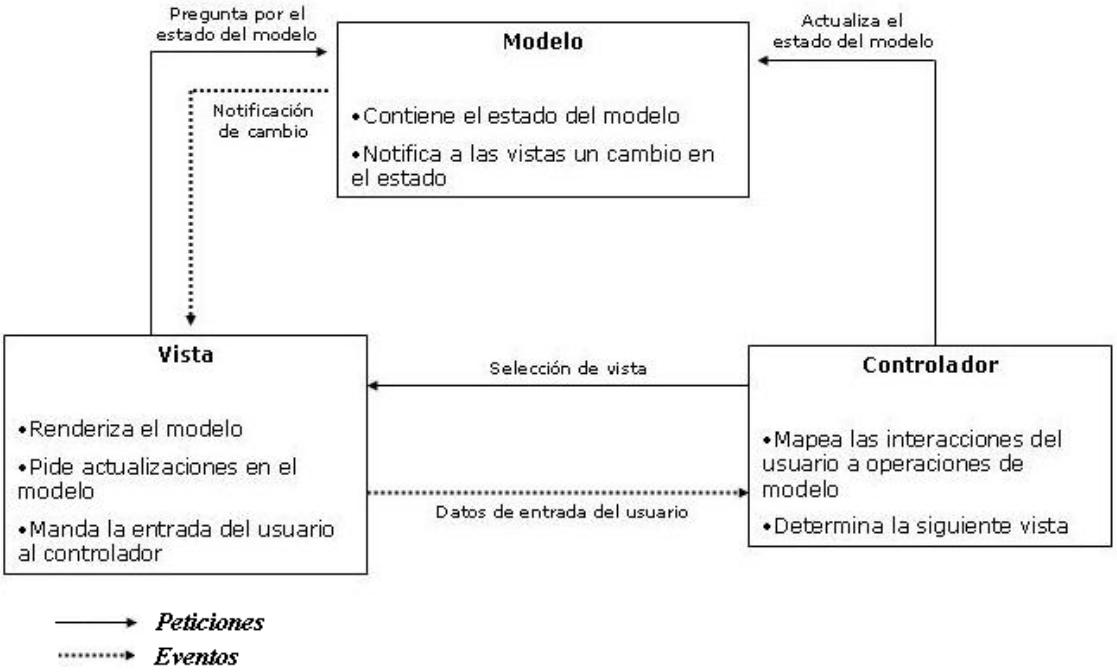


Figura 2.6 Relación entre las capas de la arquitectura MVC

El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, pulsando un botón, enlace, etc.). El controlador recibe (por parte de los objetos de la interfaz-vista) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega, frecuentemente a través de un gestor de eventos (*handler*) o *callback*. El controlador accede al modelo, actualizándolo, y posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario (por ejemplo, actualizando el carro de la compra del usuario).

A continuación, el controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se reflejan los cambios en el modelo (por ejemplo, produce un listado del contenido del carro de la compra). El modelo no debe tener conocimiento directo sobre la vista. Sin embargo, el patrón de observador puede ser

utilizado para proveer cierta indirección entre el modelo y la vista, permitiendo al modelo notificar a los interesados de cualquier cambio. Un objeto vista puede registrarse con el modelo y esperar a los cambios, pero aun así el modelo en sí mismo sigue sin saber nada de la vista. El controlador no pasa objetos de dominio (el modelo) a la vista aunque puede dar la orden a la vista para que se actualice. En algunas implementaciones la vista no tiene acceso directo al modelo, dejando que el controlador envíe los datos del modelo a la vista. Por último, la interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.

En este proyecto, la aplicación de la arquitectura MVC se fundamenta en la separación que este patrón propone en cuanto a datos, interfaz de usuario y control. El modelo estará formado por una serie de archivos de configuración XML en los que se estructura la historia que sigue el videojuego. La vista estará formada por un *MIDlet*, en el que se utilizará la API de alto nivel *Screen*, debido a las especiales características de los juegos de aventura conversacional, que hacen que sólo sea necesario mostrar por pantalla una imagen que aporta información sobre la localización del escenario, un texto descriptivo, y una serie de acciones a realizar; con lo cual se adecúa de forma bastante correcta al uso de formularios, listas y alertas que se pueden emplear usando la API de alto nivel. Para acabar, el control estará distribuido entre el servidor y el cliente, y será el encargado de trasladar las peticiones de los usuarios en la vista y actuar sobre el modelo, modificándolo, para a continuación hacer llamadas para actualizar la vista.

2.4.3 PATRÓN OBSERVADOR

El patrón observador define una dependencia del tipo uno-a-muchos entre objetos, para lo cual hay un objeto observable, objeto de datos o sujeto, cuyos cambios se van a monitorizar; y una serie de objetos observadores, que están interesados en los cambios de estado del sujeto. De manera que cuando el sujeto cambia su estado, el patrón observador se encarga de notificar este cambio a todos los objetos que han registrado su interés previamente.

El objetivo de este patrón es desacoplar la clase de los objetos clientes del sujeto, aumentando la modularidad del lenguaje, así como evitar bucles de actualización (espera activa o *polling*).

Este patrón también se conoce como el patrón de publicación-inscripción o modelo-patrón. Estos nombres sugieren las ideas básicas del patrón, que son bien sencillas: el objeto de datos contiene atributos mediante los cuales cualquier objeto observador o vista se puede suscribir a él pasándole una referencia a sí mismo. El sujeto (u objeto observable) mantiene así una lista de las referencias a sus observadores.

Los observadores a su vez están obligados a implementar unos métodos determinados mediante los cuales el sujeto es capaz de notificar a sus observadores "suscritos" los cambios que sufre para que todos ellos tengan la oportunidad de refrescar el contenido representado. De manera que cuando se produce un cambio en el sujeto, que puede haber sido realizado por alguno de los observadores, éste puede recorrer su lista de observadores avisando a cada uno de esa eventualidad.

En Java se puede heredar la funcionalidad del patrón observador de la clase *Observable* y los observadores pueden implementar la interfaz *Observer* y redefinir el método *update()*, que será el que se lance cuando hayan ocurrido cambios en el sujeto. En la Figura 2.7 se puede ver un diagrama de clases que explica el funcionamiento de este patrón:

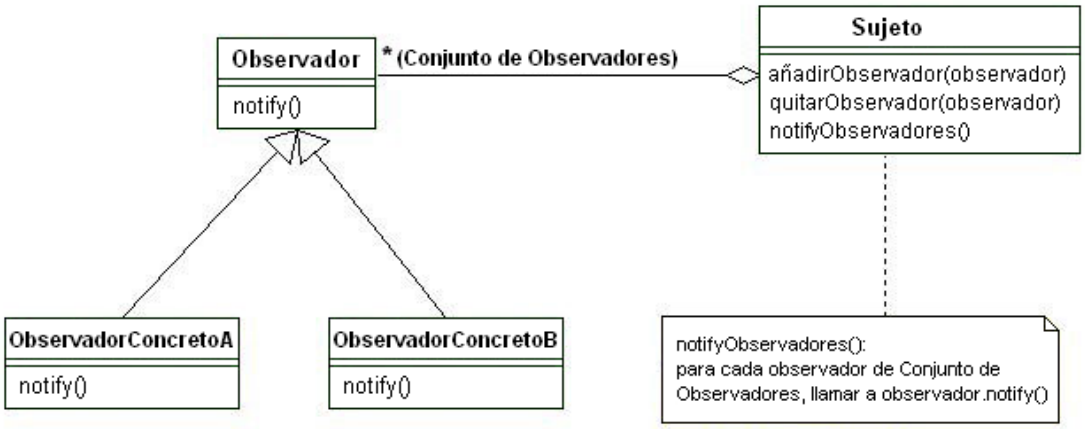


Figura 2.7 Funcionamiento del Patrón Observador

2.5 XML

XML (del inglés *eXtensible Markup Language*, Lenguaje Extensible de Etiquetas), es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C). Es una simplificación y adaptación de SGML (*Standard Generalized Markup Language*), al igual que HTML, que permite definir la gramática de lenguajes específicos. Es por ello que XML no es realmente un lenguaje en particular, sino una forma de

describir lenguajes para distintas necesidades. La diferencia con HTML es que las etiquetas XML, en lugar de presentar el contenido como hacen las etiquetas HTML, se limitan a describirlo.

En una primera aproximación, se podría decir que HTML es un subconjunto de XML especializado en presentación de documentos para la web, mientras que XML es un subconjunto de SGML especializado en la gestión de información para la web. Aunque siendo estrictos, XML contiene a HTML, pero no en su totalidad. La definición de HTML que queda contenida dentro de XML es XHTML, que por tanto queda también incluida dentro de SGML.

Al ser extensible, el lenguaje XML permite la creación de etiquetas propias para describir los datos, así como los atributos de las mismas. La única restricción a la que se debe atender es un conjunto de normas de sintaxis que están pensadas con el fin de garantizar la consistencia de los datos presentados.

Dada la organización que impone la sencilla estructura de los documentos XML, así como su flexibilidad en cuanto a la definición de etiquetas, y el hecho de que se trata de un estándar abierto; se usará para formar el sistema de archivos que componen el modelo de la arquitectura MVC implementada en este proyecto.

2.5.1 XSD

Como ya se ha comentado, la extensibilidad de los documentos XML permite definir nuevas etiquetas y atributos, que si bien es una característica muy útil, a veces puede dar demasiada libertad que puede conducir a errores. Es por ello que, en ciertas ocasiones, es necesario restringir mediante una serie de normas o reglas la estructura del documento. Para ello se puede utilizar XSD (*XML Schema Definition*), que es un lenguaje de esquema utilizado para describir la estructura y las restricciones de los contenidos de los documentos XML de forma muy precisa, yendo más allá de las normas sintácticas que impone el uso del lenguaje XML.

Los documentos esquema (con extensión *.xsd*) se concibieron como una alternativa a las DTD (*Document Type Definition*), que aportan una definición más compleja y con menor potencia expresiva a la hora de describir los documentos a nivel formal. La principal

aportación de XSD es el gran número de tipos de datos que incorpora, así como de elementos, ya que, a su vez, se trata de un documento escrito en el lenguaje XML. Por ello, XSD será una herramienta muy útil a la hora de delimitar la estructura y definir las restricciones que han de tener los archivos XML que forman parte del modelo de la aplicación.

2.5.2 PARSERS XML

Un *parser* (literalmente, analizador sintáctico) es un módulo, biblioteca o programa que se ocupa de transformar un archivo de texto en una representación interna. En el caso de XML, como el formato siempre es el mismo, no es necesaria la creación de un *parser* distinto cada vez que se desarrolla un programa, sino que es posible utilizar uno de los muchos ya existentes.

Los *parsers* suelen ser componentes muy voluminosos, con una gran cantidad de código y que necesitan elevados recursos de memoria en tiempo de ejecución. En los dispositivos MIDP, la memoria disponible para código suele ser pequeña, y las aplicaciones individuales pueden tener un tamaño máximo definido. Por tanto, es crucial contar con un *parser* que esté diseñado para ser pequeño y ligero.

No obstante, existen otras consideraciones a tener en cuenta para la elección de un parser, aunque no tan importantes como la anterior. Una de ellas es el uso de *parsers* de código abierto, opción muy interesante y atractiva debido al elevado control que se puede tener de los mismos, así como a que, dado el caso, se pueden personalizar para adaptarlos al uso que se le vayan a dar, o incluso corregir errores que pudieran tener.

Por otra parte, también hay que destacar la existencia de distintos tipos de *parsers*, así como sus características esenciales. Existen tres tipos fundamentales de *parsers*, y la elección a realizar depende de cómo se quiere que se comporte la aplicación, y del tipo de documentos que se espera analizar:

- Tipo *model*: un *model parser* lee un documento entero y crea una representación de éste en memoria. Estos parsers suelen usar muchos más recursos de memoria que cualquier otro tipo, aunque tienen la ventaja de que son más sencillos de usar, ya que una vez se carga el documento en memoria, la manipulación es sencilla.

- Tipo *push*: este tipo de *parser* lee a través de un documento entero, y, conforme encuentra varias partes del documento, notifica a un *listener*.
- Tipo *pull*: un *pull parser* lee de una vez sólo un fragmento pequeño del documento. La aplicación hace que el *parser* vaya navegando por el documento mediante repetidas peticiones de leer el siguiente fragmento.

En la Figura 2.8 se puede ver una comparativa de los *parsers* pequeños más populares en la actualidad:

NOMBRE	VERSIÓN	URL	LICENCIA	TAM.	MIDP	TIPO
kXML	1.21	http://kxml.enhydra.org/	EPL	21KB	Yes	Pull
kXML	2.1.6	http://kxml.org/	CPL	9KB	Yes	Pull
MinML	1.7	http://www.wilson.co.uk /xml/minml.htm	BSD	14KB	No	Push
NanoXML	2.2.2 lite	http://web.wanadoo.be /cyberelf/nanoxml/	zlib/libpng	6KB	No	Model
TAM	-	http://simonstl.com /projects/tam/	MPL	17KB	Yes	Push
TinyXML	0.7	http://www.gibaradunn.srac.org /tiny/	GPL	6KB	No	Model
XmlReader	-	http://kobjects.org/utils4me/	LGPL	5KB	Yes	Pull
XMLtp	1.7	http://mitglied.lycos.de/xmltp/	BSD	21KB	No	Model
Xparse-J	1.1	http://www.webreference.com /xml/tools/xparse-j.html	GPL	7KB	Yes	Model

Figura 2.8 Comparativa de *parsers* pequeños

En la columna "Tamaño", se indica el tamaño que ocupa el *parser* en un archivo *.jar*, que es una estimación del incremento de tamaño que supondría incluirlo en el *.jar* de la aplicación. La columna "MIDP" indica si el componente compilará sin necesidad de hacer cambios para adaptarlo a un entorno MIDP. Por último, en la columna "Tipo" se indica cuál es el tipo dentro del que se engloba el *parser* en cuestión.

Por lo indicado anteriormente, las consideraciones para elegir un buen *parser* adecuado a este proyecto son, en orden de importancia:

- Que sea ligero y pequeño.
- Que sea compatible con MIDP sin necesidad de modificaciones.
- Que sea software libre.
- Y por último, en condiciones similares, se elegirá siempre antes un *parser* de tipo *model* por la sencillez que aporta el tener en todo momento cargado en memoria el documento que se analiza.

Después de hacer una selección preliminar de los candidatos en base a los requisitos anteriores, quedarían kXML, XmlReader y Xparse-J. Puesto que los tres ocupan un tamaño similar en memoria, son compatibles con MIDP y tienen licencia libre; se decidió elegir a Xparse-J [18], ya que es el único que es de tipo *model*, lo cual facilitará la tarea de programación en la aplicación, manteniendo el resto de características deseables que tienen los otros candidatos.

Xparse-J aspira a ser el parser XML más pequeño del mercado. Tiene su origen en una traducción literal de Xparse, un parser XML de tan solo 5 KB escrito en JavaScript. Xparse-J lee los documentos como una cadena de texto, y presenta el documento actual como una estructura en árbol a la que se puede acceder en cualquier momento.

2.6 MODELADO UML

2.6.1 INTRODUCCIÓN

A la hora de afrontar el diseño de un sistema complejo, se puede proceder sin más que realizar un análisis previo, y lanzarse a hacer elecciones de diseño que bien pueden suponer un problema más adelante, cuando el sistema esté en un estado de construcción ya avanzado.

Es por ello que en cualquier rama de la ingeniería y la arquitectura, se ha encontrado útil la representación de los diseños de una forma gráfica. No obstante, la falta de estandarización en la manera de representar gráficamente un modelo impedía que los diseños gráficos ya realizados pudieran ser compartidos entre distintos diseñadores. Era necesario, por tanto, un lenguaje que no sólo permitiera comunicar ideas a otros diseñadores, sino que también sirviera de apoyo en los procesos de análisis de un

problema. Con este objetivo se creó el Lenguaje Unificado de Modelado (UML, del inglés *Unified Modeling Language*).

UML se ha convertido en el estándar ansiado durante tanto tiempo por profesionales de la ingeniería, permitiendo representar y modelar la información con la que se trabaja en la fase de análisis y, especialmente, en la de diseño. Es un lenguaje gráfico que proporciona una notación muy expresiva que se usa para visualizar, especificar, construir y documentar un sistema.

Como su propio nombre indica, UML es un lenguaje de modelado. Un modelo es una simplificación de la realidad. El objetivo del modelado de un sistema es capturar las partes esenciales del mismo. Para facilitar este modelado, el diseñador realiza una abstracción y la plasma en una notación gráfica, lo que se conoce como un modelado visual. Este modelado visual permite manejar la complejidad de los sistemas a analizar o diseñar. Además, el modelado visual que proporciona es independiente del lenguaje usado en la implementación, de forma que los diseños realizados utilizando UML puedan implementarse en cualquier lenguaje que soporte sus posibilidades (principalmente lenguajes orientados a objetos).

Siguiendo una definición estricta, UML es, ante todo, un lenguaje. Esto es, proporciona un vocabulario y unas reglas para permitir una comunicación. Este lenguaje proporciona las facilidades necesarias para crear y leer un modelo, pero no dice cómo crearlo, ya que de eso se han de ocupar las metodologías de desarrollo.

Además de todo lo anterior, UML es un método formal de modelado. Por lo que permite un mayor rigor en la especificación, posibilita también realizar una verificación y validación del modelo realizado, e incluso se pueden automatizar determinados procesos, de modo que se puede generar código a partir del modelo y viceversa, lo que permite que ambos estén en todo momento actualizados, con lo que se mantiene la visión de la estructura del proyecto desde el más alto nivel.

UML tiene una variedad de objetivos, aunque se pueden resumir así sus funciones:

- Visualizar. Permite expresar de una forma gráfica un sistema de forma que otro lo pueda entender.

- Especificar. Permite especificar cuáles son las características de un sistema antes de su construcción.
- Construir. A partir de los modelos especificados, se pueden construir los sistemas diseñados.
- Documentar. Los propios elementos gráficos sirven como documentación del sistema desarrollado, y pueden servir para su futura revisión.

Un modelo UML está compuesto por tres clases de bloques de construcción:

- Elementos: son abstracciones de cosas reales o ficticias (objetos, acciones, etc.).
- Relaciones: sirven para ligar los elementos entre sí.
- Diagramas: son colecciones de elementos con sus relaciones.

Por tanto, en un diagrama se tiene la representación gráfica de un conjunto de elementos con sus relaciones. Para poder representar correctamente el sistema, UML ofrece una amplia variedad de diagramas para visualizar el sistema desde distintas perspectivas. En la Figura 2.9 se pueden ver los diversos tipos de diagrama que proporciona UML:

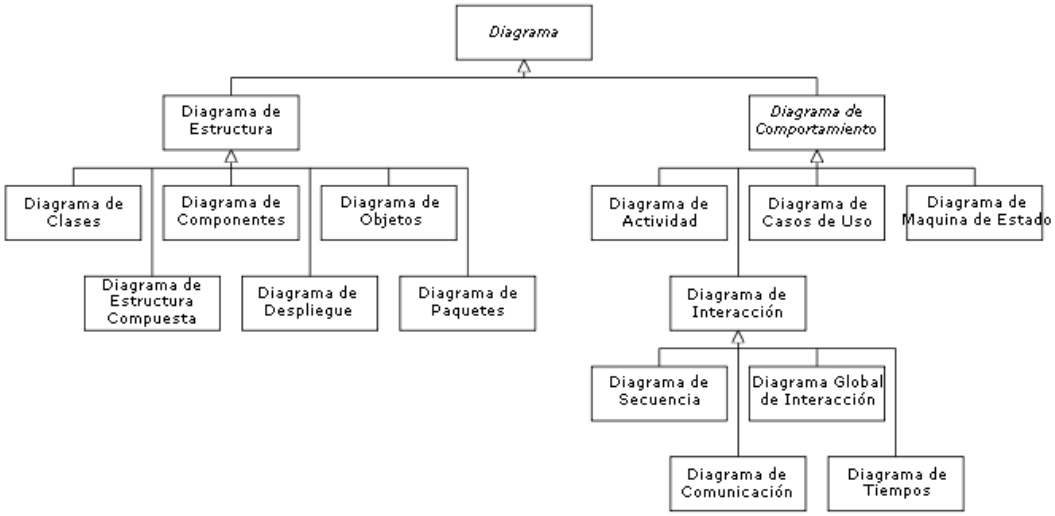


Figura 2.9 Tipos de diagramas UML

Los diagramas más interesantes (y por ello, los más usados) son los de clases, de estados y de secuencia. Debido a sus características particulares y a los diferentes puntos de vista que proponen, que se exponen a continuación, serán los diagramas que se utilizarán más adelante para realizar el diseño del motor de videojuegos que propone este proyecto.

2.6.2 DIAGRAMA DE CLASES

Un diagrama de clases muestra un conjunto de clases, interfaces y colaboraciones, así como sus relaciones. Estos diagramas son los más comunes en el modelado de sistemas orientados a objetos. Los diagramas de clases abarcan la vista de diseño estática de un sistema, como se puede ver en el ejemplo de un aeropuerto de la Figura 2.10:

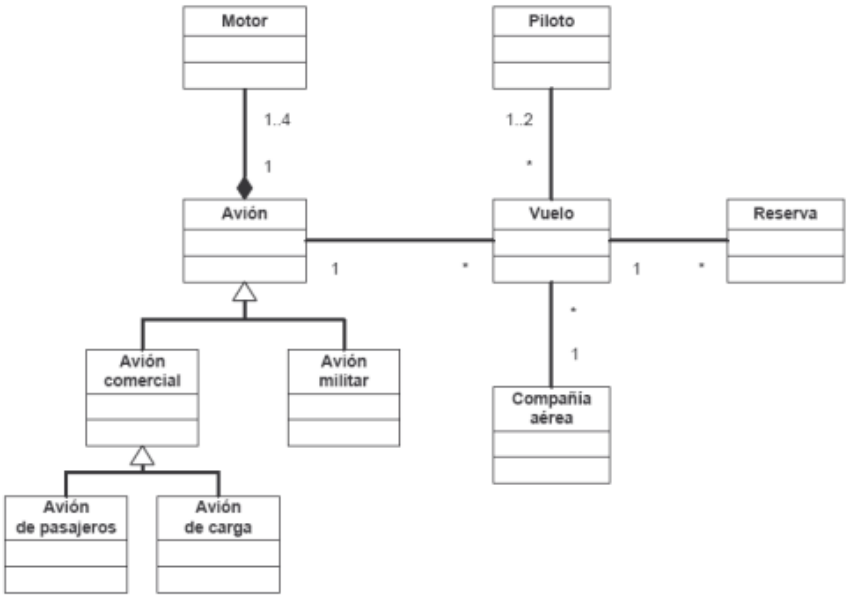


Figura 2.10 Diagrama de clases de un aeropuerto

2.6.3 DIAGRAMA DE ESTADOS

Los diagramas de estados muestran una máquina de estados, y se usan para modelar el comportamiento del sistema y resaltar el flujo de control entre estados de dicho sistema. Se pueden usar para modelar un caso de uso, una clase o un método complicado. Estos diagramas cubren los aspectos dinámicos de un sistema y son útiles para modelar la vida de un objeto, como se puede ver en la Figura 2.11.

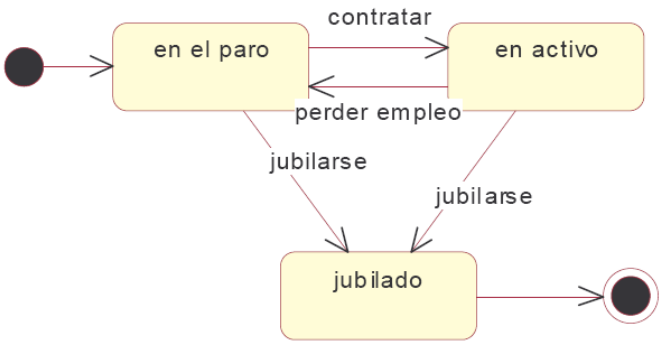


Figura 2.11 Diagrama de estados de la vida de un trabajador

En la mayoría de los casos, al utilizarse los diagramas de estados para modelar los aspectos dinámicos del sistema, esto supone modelar el comportamiento de objetos reactivos. Un objeto reactivo es aquel para el que la mejor forma de modelar su comportamiento es señalar cuál es su respuesta a eventos lanzados desde fuera de su contexto. Este tipo de objetos tienen un ciclo de vida bien definido, cuyo comportamiento se ve afectado por su pasado.

Estos diagramas también son importantes para construir sistemas ejecutables a través de ingeniería directa e inversa.

2.6.4 DIAGRAMA DE SECUENCIA

Los diagramas de secuencia son los diagramas más efectivos para modelar la interacción entre objetos en un sistema. El diagrama de secuencia muestra la interacción de un conjunto de objetos en una aplicación a través del tiempo y se modela para cada método de la clase. Este diagrama contiene detalles sobre la implementación, incluyendo los objetos y clases que se usan para implementar el escenario, así como los mensajes intercambiados entre los objetos. Estos mensajes se dibujan cronológicamente desde la parte superior del diagrama hasta la inferior. En la Figura 2.12 se puede ver un ejemplo de este tipo de diagramas.

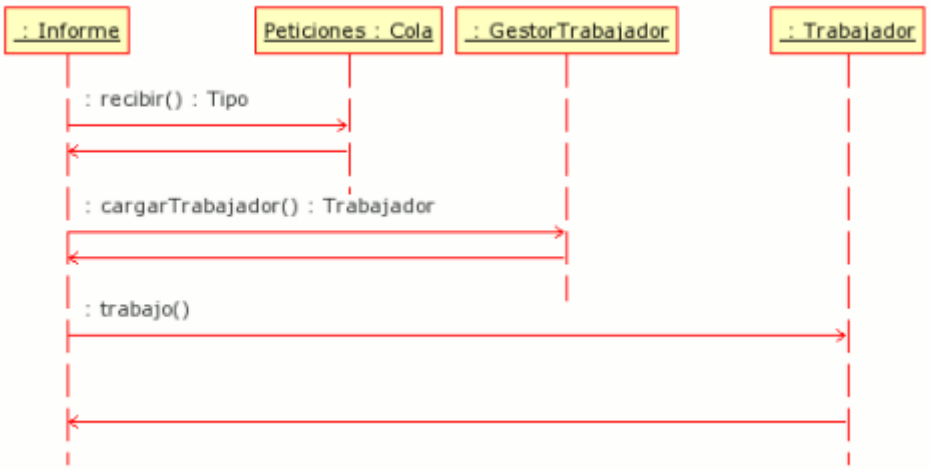


Figura 2.12 Diagrama de secuencia de un proceso de trabajo

2.7 WI-FI

Wi-Fi es un tipo de red de área local inalámbrica (WLAN, *Wireless Local Area Network*), que permite el envío y recepción de datos entre dispositivos pertenecientes a la

red sin la necesidad de cableado. En la actualidad, y debido a los cada vez más exigentes requisitos de movilidad que imponen los usuarios, así como al menor coste y facilidad de expansión que supone el no tener que realizar una instalación cableada; una gran cantidad de dispositivos incorporan capacidades Wi-Fi, entre ellos: ordenadores, consolas de videojuegos (Xbox 360, Playstation 3, Nintendo Wii, Nintendo DS), teléfonos móviles, impresoras y otros periféricos; y virtualmente todos los ordenadores portátiles de fabricación reciente.

Wi-Fi está basado en el estándar 802.11 del IEEE (*Institute of Electrical and Electronics Engineers*, instituto de Ingenieros Eléctricos y Electrónicos), del cual existen distintas variantes, que resultan en distintos tipos de Wi-Fi, cada uno con unas características que se verán a continuación:

- **802.11b**: trabaja en la banda de 2.4 GHz, sobre la que también se sitúan otras tecnologías como Bluetooth, ciertos teléfonos inalámbricos y los hornos microondas, que ocasionalmente pueden ser causa de interferencias. Tiene una tasa de datos máxima de 11 Mbps.
- **802.11a**: opera en la banda de 5 GHz, que no está tan superpoblada como la anterior. Además, utiliza un esquema de modulación diferente (OFDM, *Orthogonal Frequency Division Multiplexing*, Multiplexación por División de Frecuencias Ortogonales), que le permite mejorar al anterior estándar y lograr una tasa de transferencia máxima de 54 Mbps.
- **802.11g**: evolución que incorpora el esquema de modulación usado por 802.11a, y traslada su uso a la banda de 2.4 GHz, consiguiendo una tasa de transferencia ideal de 54 Mbps. Es el más extendido actualmente.
- **802.11n**: es un desarrollo reciente que incorpora una serie de mejoras como MIMO (*Multiple Input – Multiple Output*, Entrada Múltiple - Salida Múltiple), que hacen que logre alcanzar una tasa de hasta 600 Mbps, pudiendo operar en ambas bandas de uso libre (2.4 y 5 GHz). Sin embargo, aún es pronto para ver dispositivos en el mercado que lo implementen, ya que el estándar fue ratificado por el IEEE el 11 de Septiembre de 2009.

En uso real, nos encontramos con que la gran mayoría de puntos de acceso inalámbricos existentes en el mercado soportan 802.11b y 802.11g, y lo mismo ocurre con los dispositivos que se conectan a éstos. Por lo que se usará preferentemente 802.11g, que debido a su gran implantación, su uso más eficiente de la banda de 2.4 GHz que nos

permite tener tasas de transmisión elevadas, y su alcance en espacio libre (que puede llegar hasta los 100 metros con las antenas suministradas con el punto de acceso) será una elección adecuada en el marco de este proyecto.

La popularización del uso de este tipo de redes ha llevado a plantearse cuestiones de seguridad de las mismas, que dieron lugar a distintas técnicas que dificultan o imposibilitan el acceso no autorizado a una red inalámbrica, desde las más ingenuas como la supresión de la emisión del identificador de red SSID (*Service Set ID*), hasta las más avanzadas como el uso de cifrados WEP (*Wired Equivalent Privacy*, Privacidad Equivalente a Cableado) o WPA (*Wi-Fi Protected Access*, Acceso Protegido Wi-Fi). A fecha de la escritura de esta memoria, se considera que tanto WEP como WPA han excedido su vida útil y ya no son seguros, ya que existen vectores de ataque que pueden ser explotados para conseguir un acceso no autorizado, e incluso hay herramientas automatizadas como las que proporciona la suite WifiSlax [19] que permiten realizar estos ataques sin tener muchos conocimientos relativos a la auditoría de seguridad de redes inalámbricas. Sin embargo, WPA2, que corrige los defectos más obvios de WPA incorporando un nuevo algoritmo de cifrado AES (*Advanced Encryption Standard*, Estándar de Cifrado Avanzado), se considera aún seguro.

No obstante, y aunque la necesidad de seguridad sigue presente; en lugares como aeropuertos, hoteles, restaurantes o centros de ocio, se decide proporcionar acceso al público a sus redes inalámbricas, bien sea por el uso de redes abiertas a todo el mundo, o bien proporcionando una clave de acceso solamente a los usuarios de sus servicios, para así atraer a potenciales clientes, y mejorar la experiencia de los que ya lo son. También existe gente que, de forma desinteresada y voluntariamente, proporciona libre acceso a la red inalámbrica de su propiedad. Se estima que en ciudades como Madrid, el 35% de las redes inalámbricas existentes, están abiertas [20]. Es por ello que es fácil encontrar puntos de acceso sin clave que proporcionen una conexión a Internet, sobre todo en zonas concurridas en las que haya trasiego de público, que nos permitirán poder tener la conexión inalámbrica necesaria en este proyecto para conectarse a la aplicación.

2.8 GPRS

GPRS (*General Packet Radio Service*, Servicio General de Paquetes vía Radio), es una extensión de GSM (*Global System for Mobile communications*, Sistema Global para comunicaciones Móviles) orientada a la transmisión de datos por paquetes, que permite

velocidades de transferencia de 56 a 144 Kbps. GPRS se puede utilizar para proporcionar servicios como correo electrónico, MMS (*Multimedia Messaging System*, Sistema de Mensajería Multimedia) o Internet, ya que da un mejor rendimiento a los servicios que utilizan conmutación de paquetes, en contraposición a la conmutación de circuitos, e incluso se garantiza una cierta QoS (*Quality of Service*, Calidad de Servicio) durante la conexión.

Así, además de proveer de una velocidad de conexión mayor que la arcaica conexión de datos por GSM, GPRS tiene la ventaja de que sólo necesita reservar recursos de la red cuando se va a hacer una transferencia, por lo que su tarificación se realiza en base al volumen de tráfico generado y no al tiempo de conexión (en GSM se pagaba por segundos, tanto si se estaba aprovechando la conexión a la red como si se estaba en espera). Es decir, sólo se paga por lo que realmente se consume.

Esto abre la puerta a aplicaciones que se basan en mantenerse siempre conectadas a la red, pero que sólo hacen transacciones de datos en determinados momentos, por ejemplo sistemas de localización, que no tendrían un coste excesivo utilizando esta tecnología. Además, las compañías disponen de una serie de bonos que se pueden contratar con unos determinados megabytes de tráfico de datos [21], para poder controlar el consumo y conseguir precios más bajos que simplemente pagando por el tráfico generado.

GPRS representa un paso intermedio entre los sistemas móviles de segunda generación (2G) y los de tercera generación (3G), por lo que a menudo se le conoce también como 2.5G. Desde el punto de vista de los operadores móviles, GPRS habilita una forma sencilla de migrar a sistemas 3G en el futuro, ya que las antenas, que son los elementos más costosos de un sistema de telefonía móvil, solo sufren ligeros cambios con respecto a las de GSM, y los nuevos elementos de red necesarios para GPRS son compartidos con las redes UMTS del futuro.

GPRS es una de las alternativas que contempla el presente proyecto para realizar la conexión desde el móvil a la plataforma de videojuegos desarrollada, si bien no es la preferida, ya que es la que posee un coste más alto por megabyte y una menor velocidad de transferencia.

2.9 UMTS

UMTS (*Universal Mobile Telecommunications System*, Sistema Universal de Telecomunicaciones Móviles) es una de las tecnologías de telefonía móvil 3G, sucesora de GSM, como se comentó en el apartado 2.8. Es su sucesora ya que la tecnología GSM no podía seguir evolucionando para llegar a ofrecer servicios considerados de tercera generación, por lo que fue necesario realizar un nuevo desarrollo independiente del sistema anterior.

UMTS se pensó en inicio para su uso en dispositivos móviles, pero no está limitada a este tipo de dispositivos, pudiendo ser utilizada por otros, como por ejemplo ordenadores portátiles. Sus grandes características son las capacidades multimedia, una velocidad de acceso a Internet elevada, la cual le permite transmitir audio y vídeo en tiempo real; y una calidad de voz equiparable a la de las redes fijas. Además, dispone de una extensa variedad de servicios como acceso a Internet, videoconferencia, comercio electrónico, etc. En condiciones óptimas, puede llegar a ofrecer velocidades de 2 Mbps.

UMTS proporciona, además, un coste reducido con el objetivo de asegurar un mercado masivo de usuarios. En cuanto a la cobertura, las propias compañías ofrecen en sus webs las estimaciones de área sobre la que proporcionan servicio 3G. Se puede ver un compendio de estos mapas de cobertura en [22]. Dependiendo de la compañía, la cobertura puede llegar hasta el 90% del territorio nacional.

Como en el caso de GPRS, las compañías ofrecen una serie de tarifas o planes de datos a los usuarios [21], que les permiten pagar una determinada cantidad al mes, y poder conectarse a la velocidad máxima disponible, con un límite mensual de descarga. Si se supera ese límite, normalmente se reduce la velocidad de la conexión a una tasa inferior, limitación que dura hasta que acaba el periodo de facturación.

En el presente proyecto, se contempla UMTS como una de las posibles formas de conectar al servicio de videojuegos conversacionales desarrollado, siendo recomendada principalmente para el cada día mayor número de usuarios que ya disfrutan de una tarifa plana de datos contratada con su operador móvil [23].

2.10 ENTORNOS DE DESARROLLO (IDE)

Un entorno de desarrollo integrado o IDE (*Integrated Development Environment*) es un programa compuesto por un conjunto de herramientas que tienen como misión facilitar la labor del desarrollo a un programador. Es un entorno de programación que incluye un editor de código, un compilador, un depurador y un constructor de la interfaz gráfica. Existen IDE configurables que soportan una variedad de lenguajes de programación. La elección de un IDE adecuado puede suponer la diferencia de permitir al programador concentrarse en el desarrollo de sus aplicaciones, en lugar de tener que lidiar con las particularidades del entorno.

Siguiendo la filosofía del presente proyecto en cuanto al uso de software libre siempre que sea posible, las alternativas disponibles que permiten integrar el desarrollo de aplicaciones Java y J2ME son:

- Eclipse. Esta herramienta desarrollada originalmente por IBM, emplea módulos o *plugins* para proporcionar su funcionalidad, a diferencia de otros entornos monolíticos en los que están incluidas todas las funcionalidades, las necesite el usuario o no. Eclipse soporta de forma nativa el desarrollo de aplicaciones Java. En cuanto al desarrollo de aplicaciones J2ME, lo soporta por medio de la instalación del *plugin* EclipseME.
- NetBeans. La herramienta de Sun soporta el desarrollo de todos los tipos de aplicación Java (J2SE, J2EE, y J2ME con el paquete opcional *Mobility Pack*). Está estructurada en módulos que posibilitan extender la funcionalidad cuando sea necesario, sin tener de entrada un entorno demasiado pesado. Sun fusionó su IDE Java Studio Creator con NetBeans recientemente, por lo que ahora el IDE que recomienda utilizar en su web para desarrollo en Java es NetBeans.

Ambas plataformas conforman unos entornos de desarrollo maduros, con una gran base de usuarios que contribuyen a crear tutoriales, manuales y *plugins* para facilitar el desarrollo. Como ambos entornos se adecúan bastante bien a los requerimientos de este proyecto, la decisión entre uno u otro se basa en pequeños detalles, como por ejemplo la mayor facilidad de NetBeans para comenzar a utilizarse inmediatamente al proporcionar al usuario todo lo necesario en una única descarga, o algunos errores que se encontraron en Eclipse a la hora de añadir librerías externas a la aplicación. Es por ello que finalmente se eligió NetBeans [24] como IDE para llevar a cabo el desarrollo de la aplicación.

2.10.1 ANT

Apache Ant [25] es una herramienta para automatizar ciertas tareas repetitivas durante la fase de compilación y construcción (*build*). Está implementada en Java, y si bien se puede utilizar con otros lenguajes, despliega toda su potencia en proyectos Java. Es software libre, y se puede integrar sin problemas en los IDE anteriormente comentados.

Ant utiliza XML para describir el proceso de compilación y sus dependencias. Esto lo hace por medio de la creación de un archivo *build.xml* en el que se especifican los diferentes objetivos o *targets* que podrá llevar a cabo. Así, Ant podrá realizar tareas como la compilación de clases, compresión o empaquetado de archivos, crear o copiar directorios, etc. A través de los propios targets se pueden definir dependencias entre ellos, para darle un orden a su ejecución.

Las ventajas de Ant sobre otras alternativas para realizar este tipo de tareas es que, al estar basado en archivos de configuración XML, es multiplataforma, por lo que puede realizar las tareas que se le encarguen sobre cualquier sistema operativo. Por ejemplo, si se quiere borrar un archivo, en línea de comandos de UNIX se usa la orden “rm”, mientras que en un terminal Windows se usa “del”. Sin embargo, la diferencia de Ant es que la orden sería “<delete file=“/lib/prueba.jar”/>”, que como se puede ver es independiente del sistema operativo sobre el que se ejecuta. En la Figura 2.13 se puede ver un ejemplo de archivo *build.xml*:

```
<?xml version="1.0"?>
<project name="Hello" default="compile">
    <target name="clean" description="remove intermediate files">
        <delete dir="classes"/>
    </target>
    <target name="clobber" depends="clean" description="remove all artifact files">
        <delete file="hello.jar"/>
    </target>
    <target name="compile" description="compile the Java source code to class files">
        <mkdir dir="classes"/>
        <javac srcdir="." destdir="classes"/>
    </target>
    <target name="jar" depends="compile" description="create a Jar file for the application">
        <jar destfile="hello.jar">
            <fileset dir="classes" includes="**/*.class"/>
            <manifest>
                <attribute name="Main-Class" value="HelloProgram"/>
            </manifest>
        </jar>
    </target>
</project>
```

Figura 2.13 Contenido de un archivo *build.xml*

2.10.2 OFUSCADOR Y OPTIMIZADOR

ProGuard [26] es un ofuscador y optimizador para Java y J2ME, que tiene la ventaja de estar integrado como una parte más de NetBeans, aunque también puede usarse por separado. Un ofuscador es un software al que se le proporciona un código desarrollado, tras lo que se encarga de detectar y eliminar clases, campos, métodos y atributos no usados en dicho código. Hecho esto, realiza una optimización de los *bytecodes* de los métodos. Por último, se realiza el paso de la ofuscación, que se encarga de renombrar las clases, campos y métodos restantes utilizando nombres cortos sin significado, para ahorrar espacio.

El resultado de este proceso es una base de código más pequeña y eficiente, con el efecto añadido de que hace que sea más difícil aplicar ingeniería inversa a los programas sobre los que se ha aplicado la ofuscación. Cualquier software comercial que se precie lleva aplicado un proceso de ofuscación y optimización.

En este proyecto, se busca la ventaja que aporta el ofuscador y optimizador en cuanto a la reducción del tamaño del código desarrollado, y no tanto las dificultades que añade ante posibles intentos de ingeniería inversa. Al tratarse de una aplicación que se va a desplegar sobre terminales móviles, que son aparatos con recursos limitados, es interesante aprovechar cualquier forma que haya de reducir el tamaño del código y el consumo de memoria, siempre que la aplicación final no se vea afectada en el proceso.

2.11 EMULADORES

Durante el desarrollo y depuración de este proyecto, se han usado distintos emuladores para facilitar la tarea de localización y corrección de problemas. Sin la ayuda de estos emuladores, el desarrollo habría sido más lento, y habría tenido una complejidad mucho mayor.

En este epígrafe se mencionan los emuladores que se usaron para llevar a cabo las pruebas del *MIDlet*, así como sus características y el ámbito en que se utilizaron:

- **Sun Wireless Toolkit** [27]: Este emulador viene proporcionado por Sun dentro del paquete en el que se distribuyen las herramientas necesarias para desarrollar aplicaciones MIDP sobre la configuración CLDC. Al estar desarrollado por Sun

en base a las especificaciones de MIDP, sin tener en cuenta las particularidades que se dan en los terminales físicos reales, este emulador tiene su principal utilidad en la fase inicial del desarrollo, pero no se debe considerar como fiable por completo, ya que puede ejecutar una aplicación sin ningún error, y sin embargo no está garantizado el correcto funcionamiento de ésta más tarde cuando se instala en un teléfono móvil. Por otra parte, una característica de este emulador es el poder ejecutar múltiples instancias sobre la misma máquina, por lo que fue de una gran ayuda para realizar las pruebas en las que fue necesario implicar a más de un terminal, como es el caso del multijugador. Por último, cabe destacar que su consumo de memoria es reducido, lo que facilita también la ejecución de múltiples instancias y la rapidez en la corrección de errores. Se puede ver una imagen de este emulador en la Figura 2.14, a la izquierda.

- **Nokia S60 SDK** [28]: Este es el emulador que proporciona Nokia para el desarrollo sobre sus teléfonos móviles de la Serie 60. Está basado en el sistema operativo Symbian, que es una de las plataformas más usadas en la actualidad para dispositivos móviles, no sólo por Nokia sino también por otros fabricantes como Sony Ericsson, Motorola o Samsung. Este emulador tiene la particularidad de emular de una forma muy fidedigna el comportamiento de los teléfonos de la Serie 60 de Nokia, por lo que una aplicación que se ejecute de forma correcta en él es muy probable que lo haga de igual forma al instalarse en un dispositivo móvil real. La contrapartida que tiene esto es que, para emular un teléfono de la Serie 60, hacen falta unos requisitos considerables de memoria, así como una capacidad de proceso que no es despreciable, por lo que puede transcurrir un lapso de tiempo apreciable desde que se lanza el emulador hasta que queda en estado funcional, dispuesto a ejecutar la aplicación a probar, siendo esto un impedimento para llevar a cabo una labor rápida de testeo. Además, se encuentra limitada la ejecución de más de una instancia del emulador sobre una misma máquina, por lo que no es adecuado para, por ejemplo, las pruebas relativas a multijugador en la aplicación desarrollada. Sin embargo, a la hora de comportarse como un teléfono móvil real, no tiene rival, por lo que su aportación a la corrección de problemas también fue considerable. Se puede ver una imagen de este emulador en la Figura 2.14, a la derecha.

Figura 2.14 Los emuladores Sun WTK y Nokia S60

2.12 ANALIZADOR DE TRÁFICO

Durante el diseño de una aplicación que tiene como parte principal la conexión a la red, como en el caso de este proyecto, es necesario tener una forma de poder realizar distintas comprobaciones y estadísticas en cuanto a la conexión. Es en ese momento cuando entran en juego los analizadores de red, que son programas para realizar análisis y solucionar problemas en el desarrollo de aplicaciones que disponen de conexión a la red.

Dentro de los analizadores de tráfico, y una vez más desde el prisma de la búsqueda de una herramienta que cumpla con el cometido que se busca y, a ser posible, sea software libre, se encuentra la herramienta WireShark [29] (anteriormente conocida como Ethereal). WireShark es un analizador de protocolos con interfaz gráfico, que permite examinar datos de una conexión viva o de un archivo de captura anteriormente guardado en disco. Además dispone de una potente capacidad de filtrado, que hace que navegar por la información capturada, que puede ser de dimensiones considerables, sea más sencillo.

Sin embargo, la potencia de WireShark no estriba únicamente en su capacidad de análisis de datos y realización de estadísticas, sino en que es un software que entiende la estructura de un gran número de protocolos de red. Así, puede mostrar la encapsulación y los campos junto a lo que significan los distintos paquetes especificados por una serie de protocolos.

Los datos se pueden capturar tanto de una conexión por cable (Ethernet), como de una conexión Wi-Fi, entre otros. Por ello, será útil para ver qué cantidad de tráfico utiliza la aplicación móvil, y qué forma tiene este tráfico.

CAPÍTULO 3: DISEÑO DE LA APLICACIÓN

3.1 INTRODUCCIÓN

En el presente capítulo se detallan las distintas partes que forman la aplicación desarrollada, así como las relaciones existentes entre ellas. Además, se verán los procesos que se llevan a cabo en el sistema. Por ello, este capítulo está dividido en tres partes. En la primera parte se expondrá una visión general de la aplicación para tener una idea sobre los componentes que la forman y sus relaciones a alto nivel. A continuación, en la segunda parte, se analizará el funcionamiento de la aplicación, así como de los procesos en los que intervienen sus diversas partes y que son los que, en última instancia, proporcionan la funcionalidad al usuario. Por último, se desgranará el funcionamiento de cada uno de los componentes, entrando a fondo en la estructura de la aplicación, de la que se detallará el diseño y el funcionamiento de cada una de las partes que la componen.

3.2 VISIÓN GENERAL

La aplicación desarrollada es un “servicio de videojuegos conversacionales multijugador en red para dispositivos móviles”. Es decir, ante todo, proporciona un servicio de videojuegos conversacionales. Para ello, se modifica y amplía un motor de videojuegos conversacionales ya existente. Un motor es una aplicación genérica que produce un efecto cuando se le proporciona una alimentación. En este caso, el motor es válido únicamente para ejecutar un tipo de videojuegos, las aventuras conversacionales, que tienen una serie de características y forma de desarrollar la historia determinadas. El servicio de videojuegos conversacionales propuesto es multijugador, de modo que permite que participen en el juego varios usuarios de forma simultánea. Además, es en red, por lo que esta funcionalidad multijugador se ofrece a través de una red formada por la aplicación a través de Internet. Finalmente, el servicio está enfocado a dispositivos móviles, para lo que se tienen en cuenta sus particulares características de espacio reducido en pantalla y limitada capacidad de procesamiento.

La finalidad de la aplicación desarrollada es posibilitar la creación de este tipo de juegos de forma relativamente sencilla y sin que para ello sean necesarios conocimientos de programación. De esta forma, es necesaria una estricta organización de la estructura de estos videojuegos, para que sea fácil incorporar nuevos títulos al motor. Para ello se ha seguido la arquitectura MVC en el desarrollo, según la cual se establece una separación entre las tres partes que le dan nombre: Modelo, Vista y Control.

Así, se tendrá un sistema de archivos XML, que conforman el modelo de la aplicación, y que son los que incluyen la historia, los personajes que participan en ella, los objetos que se pueden encontrar y las acciones a realizar en el juego; así como la personalización para cada uno de los usuarios.

Por su parte, la vista de la aplicación será presentada por un *MIDlet* que reside en el dispositivo móvil del jugador y que se encargará de adaptar los contenidos a mostrar en el teléfono. Estos contenidos se envían desde el servidor hacia el cliente por medio de sendos módulos de comunicaciones que hacen de interfaz y proporcionan una conexión entre ambos. Una vez establecida inicialmente esa conexión, se despliega sobre ella un protocolo desarrollado a modo de máquina de estados, que hace que tanto cliente como servidor sigan un determinado flujo de programa que les permita en cada momento realizar las acciones que el jugador requiera, ya sea coger un objeto, registrar un nombre de usuario, crear una nueva partida multijugador, etc. Asimismo se establece una segunda conexión del servidor hacia el cliente que se usa para mandar actualizaciones asíncronas (por ejemplo, cuando una localización tiene un límite de tiempo establecido y el servidor avisa de que este tiempo ha expirado), y en el caso de que se esté en una partida multijugador, para mantener en todo momento la coherencia del modelo.

Por último, en el servidor reside la parte del control de la aplicación, que se encarga de implementar la inteligencia necesaria para saber en todo momento los contenidos que ha de servir a los jugadores, cómo ha de convertir la información aportada por el cliente móvil en una petición de cambio al modelo y qué serie de consecuencias se producirán, en su caso. En el caso de una partida multijugador también permite que se envíen mensajes a otros jugadores de la partida, y permite que los cambios iniciados por un jugador en el modelo se propaguen de forma automática al resto de jugadores de la misma partida, de modo que todos los jugadores dispongan en todo momento de la versión actualizada del modelo, garantizando así su coherencia.

En la Figura 3.1 se puede ver un diagrama de la arquitectura de la aplicación:

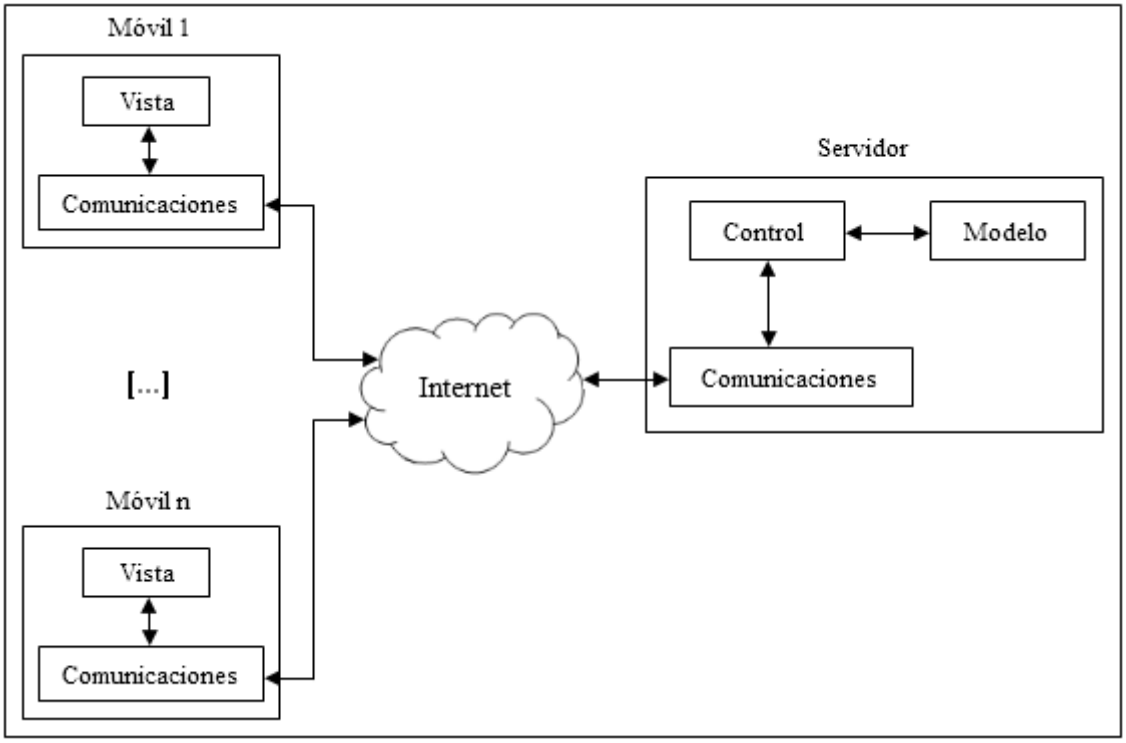


Figura 3.1 Arquitectura de la aplicación

3.3 FUNCIONAMIENTO DE LA APLICACIÓN

En este apartado se va a comentar y explicar el funcionamiento global de la aplicación. Para ello, aún no se profundizará en los detalles, sino que se expondrá, desde un punto de vista funcional qué es lo que hace la aplicación. Una discusión sobre los detalles de la implementación se encuentra más adelante en el apartado 3.4.

En la Figura 3.2 se puede ver un diagrama de estados de la aplicación, en el que se detallan los procesos que tienen lugar en ella, cada uno de los cuales implicará a uno o más métodos de una o más clases distintas.

En primer lugar, se muestra al usuario la pantalla de bienvenida de la aplicación. Desde aquí se le permite configurar y proceder con la conexión a la plataforma para ejecutar un juego determinado entre los que están disponibles.

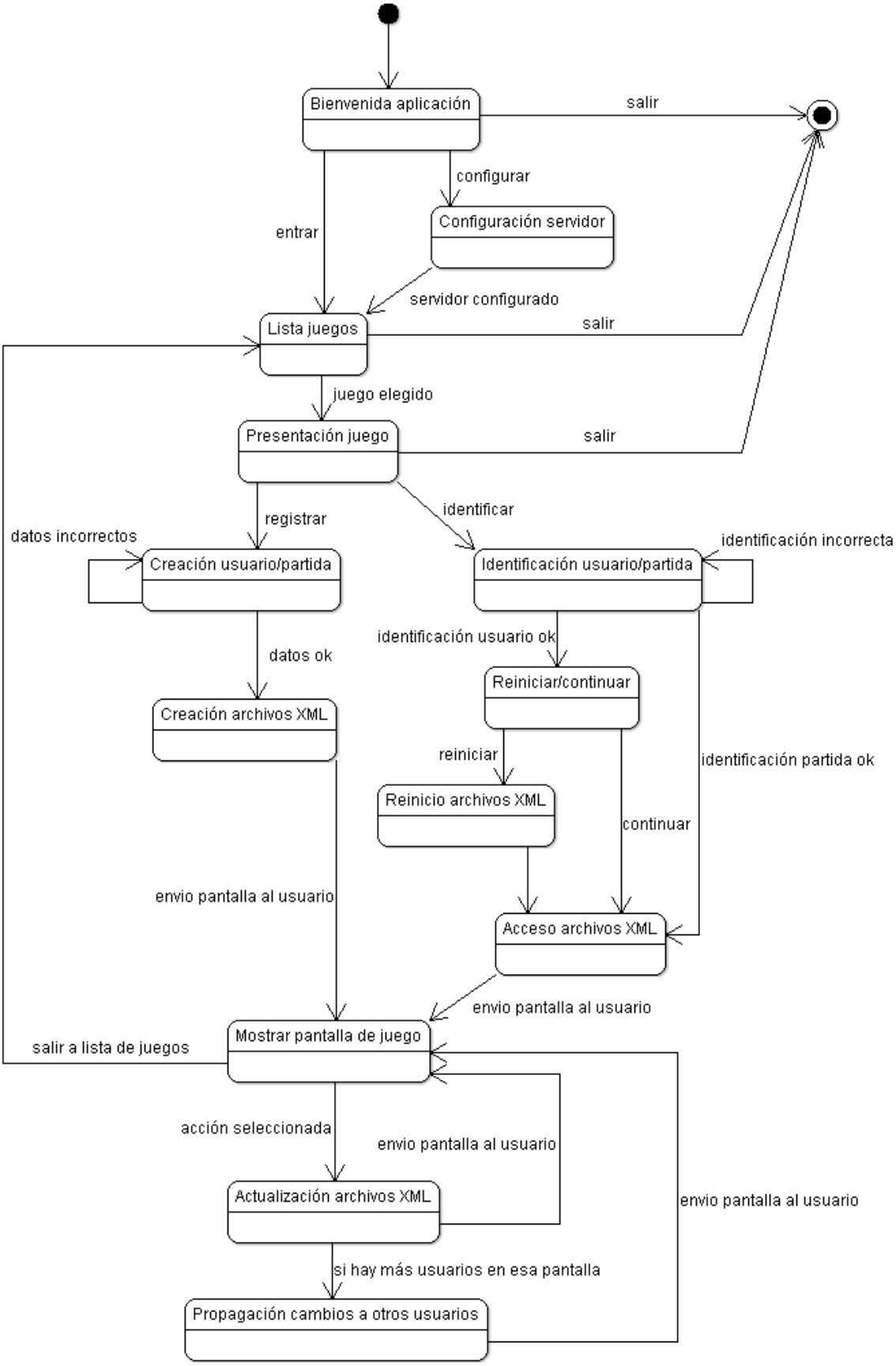


Figura 3.2 Diagrama de estados de la aplicación

A continuación, el usuario se encuentra con la posibilidad de realizar un nuevo registro de una partida individual o colectiva, o bien, de identificarse y seguir jugando en una partida individual o colectiva ya comenzada anteriormente.

Por último, llegados a este punto el usuario ya se encuentra dentro de una partida (sea ésta individual o colectiva, de nueva creación o ya existente), por lo que se entra en la ejecución del bucle de juego como tal, con el envío que realiza el servidor de localizaciones al cliente, la elección de acciones por parte de éste, que propicia el envío de una nueva localización para continuar con el flujo de juego.

3.3.1 PROCESOS DE LA APLICACIÓN

En este apartado se explicará en más detalle el funcionamiento de los procesos que lleva a cabo la aplicación desarrollada.

3.3.1.1 Conexión al servicio y presentación del juego

Este proceso es el primero en tener lugar cronológicamente hablando, ya que se ejecuta en cuanto el usuario lanza la aplicación en el móvil. La primera parte de este proceso se lleva a cabo sin realizar aún ningún acceso a la red, ya que para ello es necesario configurar previamente la conexión al servicio.

Al lanzar la aplicación, se tendrá una imagen general que ilustra el motor desarrollado a modo de carátula, así como las opciones de entrar a la plataforma o configurar los datos de conexión al servidor. Si es la primera vez que el usuario se conecta al motor, se le obligará a introducir los datos de conexión con el servidor, es decir, la dirección IP de éste y el puerto en el que está a la escucha inicialmente. Si no, se utilizarán para la conexión los valores que se usaron en la última sesión de juego.

Cabe reseñar que el servidor utilizará dos puertos para comunicarse con los clientes, que tendrán que ser consecutivos. Inicialmente se establecerá una conexión utilizando el primer puerto, por la que se enviará la información de identificación de usuario, así como los distintos formularios usados en el proceso de identificación o registro. Cuando se haya recibido por parte del usuario la primera pantalla de juego, se creará otra conexión usando el segundo puerto, por medio de la cual se recibirán actualizaciones asíncronas del servidor y, si se está en una partida multijugador, actualizaciones provocadas por otros usuarios.

Llegados a este punto, una vez configurada la IP y puerto del servidor, se realiza la primera conexión con éste y, si tiene éxito, se mostrará un listado de los juegos disponibles en la plataforma en ese momento. Para recopilar esta lista, el servidor simplemente hace un listado de todos los directorios que se encuentran dentro del directorio de juegos de la plataforma, por lo que al añadir un juego a la estructura de directorios en el servidor no será necesaria ninguna configuración adicional para que el nuevo título se muestre en la lista de los videojuegos disponibles. Ni siquiera será necesario un reinicio del servidor.

Hay una serie de errores de conexión que pueden ocurrir, y que han sido contemplados para que el usuario pueda seguir utilizando la aplicación y ésta se recupere del error. Así, los distintos tipos de error de conexión que pueden ocurrir se agrupan en los siguientes:

- La dirección IP del servidor introducida no es una IP válida. En el cliente se comprueba que dicha IP esté compuesta únicamente por números y puntos, que haya exactamente tres puntos y cuatro números en configuración número – punto – número – punto – número – punto – número, y que los números estén entre 0 y 255. Si no se cumplen estas condiciones, ni siquiera se intenta conectar, y se muestra al usuario que ha habido un error en la introducción de la IP del servidor, por lo que se le invita a corregirla.
- El puerto del servidor introducido no es un puerto válido. En el cliente de nuevo, se comprueba que el puerto introducido sea un número, y que este número esté entre 1 y 65535. Si no es así, no se trata de un puerto TCP válido, por lo que, como en el caso anterior, no se intenta conectar y se muestra al usuario un aviso de error indicándole que el puerto introducido no es válido, acto seguido se le invita a introducir otro puerto válido.
- La dirección IP y puerto introducidos, no coinciden con los del servidor de la plataforma de videojuegos. En este caso, se intenta la conexión, y si no se ha conseguido pasado un período de *timeout*, se considera que los datos de la conexión son erróneos, o bien que el servidor no está disponible en ese momento.
- Una vez conectados a la plataforma, si por cualquier motivo se pierde la conexión con el punto de acceso Wi-Fi, con la red GPRS o UMTS, o el servidor cae; se detecta automáticamente que ha ocurrido un error en la conexión.

En estos dos últimos casos, ocurrido el error, se presentará al usuario la posibilidad de intentar reconectar por si el problema fuese transitorio, cambiar los datos de conexión con el servidor en el caso de que se haya cometido un error al introducirlos, o bien abandonar

la aplicación si los problemas no se resuelven. Si se intenta la reconexión, tanto cambiando los datos de configuración del servidor como sin haberlos cambiado, y sigue sin tener éxito, se vuelven a mostrar estas opciones al usuario.

Una vez solucionados los problemas, o si éstos no han tenido lugar; el usuario puede decidir abandonar la aplicación o elegir un juego de los mostrados en la lista. Si selecciona uno de los juegos, se accederá de nuevo al servidor y se le pedirán los contenidos necesarios para mostrar la pantalla de presentación del juego.

La pantalla de presentación del juego está formada por una imagen que ilustra su contenido, así como un texto introductorio sobre el videojuego en cuestión. Además, se incluye la opción de abandonar la aplicación o bien elegir tomar parte en una partida individual (un jugador) o colectiva (multijugador).

3.3.1.2 Registro del usuario/partida

Este proceso tiene lugar en el caso de que se quiera dar de alta una nueva partida, vaya a participar en ésta un único jugador o varios. Si se quiere registrar una nueva partida individual, se presenta al usuario un formulario en el que se le pide que elija e introduzca un nombre de usuario, que se utilizará para guardar los progresos en el juego, y que establezca una contraseña para proteger su cuenta de accesos no autorizados. Para establecer la contraseña es necesario introducirla en dos campos y que estos coincidan, para proporcionar la seguridad de que el jugador no se haya equivocado al introducirla y más tarde le sea imposible volver a acceder a sus progresos guardados. Los nombres de usuario elegidos serán locales al juego, por lo que se podrá tener un mismo nombre de usuario correspondiente a dos jugadores distintos en juegos distintos, para así poder ofrecer una mayor cantidad de nombres disponibles.

En el caso de registro de una nueva partida multijugador, se pedirá al usuario que elija e introduzca un nombre de partida, junto con una contraseña que tendrá que introducir dos veces para comprobar que no ha habido errores. Además del nombre de partida y la contraseña, que tendrán que ser conocidas por todos los jugadores que quieran entrar a formar parte de dicha partida más adelante, también se pide al creador de la partida que introduzca un nombre de usuario que sirva para identificarle dentro del juego. Aquí, se tendrá que los nombres de usuario son locales a la partida, para poder ofrecer la posibilidad

de una mayor flexibilidad a los jugadores, que dentro de una partida puedan adoptar el alias que deseen.

En ambos casos, se pueden producir diversos errores en el proceso. Los errores que pueden ocurrir son los siguientes:

- Se ha dejado algún campo vacío en el formulario, por lo que se informa al usuario de que ha de rellenar todos los campos para poder continuar con el proceso.
- La contraseña introducida en ambos campos no coincide, por lo que se informa al usuario de que debe probar a introducirla de nuevo.
- Se está intentando registrar un nombre de usuario o partida que ya existe. Por tanto, se informa al usuario de esa eventualidad y se le pide que elija otro nombre para continuar.

En todos estos casos se informa al usuario del error ocurrido por medio de una alerta en el móvil. Sólo en el último caso será necesario acceder al servidor para realizar la comprobación, en los dos casos anteriores la verificación se realiza de lado del cliente.

Si el proceso ha transcurrido sin errores, o si los ha habido pero se han solventado, en este punto ya se ha realizado la creación de los archivos de configuración personalizados para el usuario/partida en el servidor, y estarán listos para comenzar con la ejecución del juego. Es necesario recalcar que, en el caso de una partida multijugador, todos los usuarios que se encuentran en una misma partida comparten un único grupo de archivos de configuración, que indicará la situación y progreso en el juego de todos ellos.

3.3.1.3 Identificación del usuario/partida

Este proceso tiene lugar si se ha decidido continuar con una partida individual ya comenzada, o si se quiere entrar a formar parte de una partida multijugador que ya está en juego.

Usuario individual

En este caso, se le pedirá al jugador que introduzca su nombre de usuario y contraseña, para poder identificarse. Se pueden dar los siguientes problemas:

- Se ha dejado algún campo vacío en el formulario. Como ya se ha comentado, se le pide al usuario que rellene todos los campos para continuar.

- El nombre de usuario introducido no existe. En este caso, el servidor comprueba que no existe dicho usuario en el directorio de usuarios del videojuego seleccionado, e informa de lo ocurrido.
- La contraseña introducida no se corresponde con el nombre de usuario introducido. Aquí el problema es que el usuario existe, pero la contraseña definida no es la que se ha introducido, por lo que se permitirá al usuario volver a intentarlo.

De nuevo, se informa al usuario de los errores ocurridos por medio de una alerta en el terminal. El usuario puede leer la descripción del error y pulsar aceptar para continuar y corregir los datos que correspondan.

Una vez realizado el proceso anterior, se presentarán dos opciones al usuario: reiniciar y poner a cero los avances que haya efectuado previamente ese jugador, útil, por ejemplo, si se encuentra perdido en el juego y prefiere comenzar de nuevo. Y también se presentará la opción de continuar jugando por el punto en que se dejó la historia por última vez. En cualquiera de los casos, tras elegir una de las dos opciones, se vuelven a crear los archivos de configuración, o se recuperan los ya existentes, dejándolos listos para comenzar la ejecución del juego.

Partida multijugador

El jugador llega a este punto si desea llevar a cabo una de las dos acciones siguientes:

- Entrar a formar parte de una partida multijugador que ya existe, pero en la que no ha entrado previamente.
- Entrar a formar parte de una partida multijugador en la que ya ha participado anteriormente.

En el primer caso, se añade al usuario a la lista de jugadores que están participando en la partida, modificando los archivos de configuración comunes a todos los miembros de ésta. En el segundo, se consulta el lugar donde estaba cuando se guardaron por última vez sus progresos, y su estado (objetos que llevaba en su inventario, acciones realizadas, etc.), para poder recuperarlo.

En el caso de que el jugador ya haya estado anteriormente en la partida, entra automáticamente en el punto en el que se quedó la última vez que jugó, ya que la versión

multijugador de la aplicación está enfocada a crear un mundo persistente en el que los personajes existan independientemente de que estén jugando o no.

En cuanto a los problemas que pueden ocurrir durante la fase de identificación de una partida multijugador, se pueden resumir en los siguientes:

- Se ha dejado algún campo vacío en el formulario, por lo que se informa al usuario de esta eventualidad y se le pide que los rellene todos para continuar.
- El nombre de partida introducido no existe. En este caso, el servidor comprueba que no existe dicha partida en el directorio de partidas multijugador del videojuego seleccionado, e informa de lo ocurrido.
- La contraseña introducida no se corresponde con el nombre de partida introducido. Aquí el problema es que la partida existe, pero la contraseña que se ha introducido no es la que da acceso a dicha partida, por lo que se informará al usuario y se le dará la opción de volver a intentarlo.

Una vez superada esta fase, al igual que en el caso de partida individual, se tienen los archivos de configuración ya preparados para iniciar el proceso de juego, que se comenta a continuación.

3.3.1.4 Ejecución del videojuego

Este proceso se inicia cuando ya se han cargado los archivos de configuración del modelo, y están preparados para jugar. Por tanto, el servidor procede a realizar el envío del contexto del juego en el que se encuentra el usuario.

Una vez recibido, el cliente muestra la imagen que ilustra la localización junto con una descripción de la misma, así como una lista de posibles acciones a realizar. Cuando el usuario decida realizar una acción, se envía al servidor una petición con dicha acción junto con las opciones que la acompañan (con qué personajes u objetos se realiza, en qué localización, etc.). El servidor intentará entonces ejecutar la acción pedida, para lo que consultará los archivos de configuración del usuario, mirando si es posible hacerlo. Si se puede ejecutar, lo hará modificando de la manera oportuna los archivos correspondientes (por ejemplo, si el usuario deseaba moverse de localización y esto es posible, el servidor modificará en el archivo *user.xml* la localización del usuario, haciendo que apunte al nuevo lugar) y procesando después las consecuencias que se pudieran producir. Si no, se mostrará

un mensaje de error al usuario por medio de una alerta en el que se le informe de la imposibilidad de realizar la acción, y no se cambiarán los archivos de usuario.

Tras el envío de la primera localización al usuario, se establecerá una segunda conexión usando el segundo del par de puertos que va a utilizar la plataforma. Por medio de esta conexión, el cliente se mantiene a la escucha esperando a que lleguen notificaciones de actualizaciones asíncronas del servidor o, en caso de que se esté en una partida multijugador, notificaciones de que ha habido cambios en el modelo o en el entorno de juego (por ejemplo, usuarios que entran o salen de la localización en la que se está). Se comienza a escuchar para atender a este tipo de eventos en este momento porque sólo tiene sentido empezar a hacerlo cuando se entra en el juego propiamente dicho.

Si se da el caso de que ha habido un cambio en el entorno de juego o en los archivos de configuración, se envía el nuevo contexto de juego al usuario que lo provocó en el caso de que se trate de un único jugador, o de que no haya ningún usuario más en esa pantalla. Si hay más usuarios en la pantalla en la que se realizaron los cambios, se les notificará también a ellos de que ha habido modificaciones producidas en esa localización por otro usuario, para que los clientes correspondientes realicen automáticamente una petición de actualización del contexto de juego al servidor. Por lo que se actualizará dicha localización para todos los jugadores que estén en ella trayendo de nuevo el contexto de juego desde el servidor a estos clientes.

Hay un caso especial de notificaciones enviadas de forma asíncrona del servidor al cliente, que tiene la particularidad de ser el único que se da en partidas individuales. Este caso es que la localización tenga un tiempo máximo definido antes de que ocurra un evento. Por ejemplo en el caso de que un jugador llegue a una localización al borde de un precipicio, puede ocurrir que, pasado cierto tiempo en esa localización, se produzca un desprendimiento y el jugador caiga. Para ello, en determinado tipo de localizaciones, se establece un temporizador en el servidor desde el momento en que llega un usuario. Este temporizador es local a dicho usuario. Y cuando ha pasado un cierto tiempo definido, puede notificársele de que ha ocurrido un determinado evento. Puede haber definidos varios temporizadores en una misma localización, que vayan expirando uno tras otro (por ejemplo, para avisar al usuario de que el suelo del precipicio empieza a resquebrajarse, antes de hacer que caiga directamente).

En cuanto a las acciones a realizar por los jugadores, se detalla a continuación el funcionamiento de algunas de ellas, que tienen la particularidad de poder provocar un reenvío del contexto de juego a otros jugadores si la partida es multijugador.

Dar a

Esta acción se usa en juegos individuales para que el jugador pueda entregarle objetos a los NPC. En el caso multijugador, se puede realizar esta acción con otro jugador, de modo que un objeto que posea un jugador en su inventario, pasa a manos del otro jugador. En este caso, se realiza una petición de envío del nuevo contexto de juego en el jugador que recibe el objeto, para que se actualice su inventario con el nuevo objeto disponible.

Hablar con

Cuando se está en una partida individual, esta opción sólo es aplicable a los NPC, con los que se puede hablar siguiendo una conversación preestablecida dentro de un guión. Sin embargo, en un contexto multijugador, tiene sentido querer comunicarse con otros jugadores humanos en la partida, y esta comunicación no tiene por qué tener impuesta ninguna limitación sobre el tipo de frases que se pueden decir. Es por ello que si se elige la acción “hablar con” aplicada sobre otro jugador, aparece una caja de texto en la que se puede escribir un mensaje que llegaría al destinatario. Para ello, como en el caso anterior, cuando el servidor detecta que hay un envío de mensaje a un usuario, provoca un reenvío del contexto de juego para que el receptor pueda ver el mensaje enviado por el otro jugador. También será posible, si hay más de dos jugadores en una localización, hablar con todos ellos simultáneamente por medio de una opción que aparece dentro del menú que se presenta al elegir “hablar con”. Esta opción se mostrará como “hablar a todos”, y solamente aparecerá en el caso de que haya más de dos jugadores en una misma pantalla al mismo tiempo.

Otras acciones en contexto multijugador

También hay que reseñar que hay otras acciones que provocan un reenvío del contexto de juego a los jugadores que comparten escenario con el jugador que realiza los cambios. Son lo que se han llamado hasta ahora cambios en el entorno de juego. Por ejemplo, si un jugador sale de una pantalla, se notifica a todos los jugadores que estaban en ella, para que sepan que ya no podrán hablar con él ni darle objetos. Asimismo, si un jugador entra a una pantalla, también se notifica a todos los jugadores que están en ella, para que lo incorporen a la lista de usuarios con los que se puede interactuar.

3.4 ESTRUCTURA DE LA APLICACIÓN

3.4.1 SISTEMA DE ARCHIVOS

3.4.1.1 Arquitectura del sistema de archivos

Este proyecto está desarrollado siguiendo la arquitectura MVC, como ya se ha comentado con anterioridad. Por tanto, será necesario disponer de una estructura de directorios ya definida, que albergue en su interior los archivos y directorios necesarios para definir y configurar todos los videojuegos que se vayan a implementar para la plataforma. Esta estructura tiene que existir para que el motor conozca los lugares en los que tiene que buscar los recursos necesarios para servir los contenidos a los usuarios, así como para permitir la sencilla incorporación de nuevos juegos a la plataforma.

En la Figura 3.3 se puede ver un esquema de la estructura típica de archivos del modelo. Cualquier videojuego que se desarrolle para ser ejecutado en este motor, tendrá que integrarse dentro de esta estructura. Se puede ver que todo el sistema está contenido en un directorio raíz de nombre “data“, dentro del cual se encuentran una serie de directorios, cada uno de los cuales implementará un videojuego distinto.

Una vez dentro del directorio correspondiente a un juego determinado, hay tres directorios más. En el directorio “XSD” se encuentran los archivos Schema (*.xsd*) utilizados en la creación de los archivos XML del modelo, que validan el contenido de los mismos y facilitan su creación, indicando si la estructura utilizada es errónea. También se encuentran dos directorios más, “images”, que como su nombre indica, albergará las imágenes utilizadas para ilustrar cada escenario, así como las pantallas de presentación y despedida; y “files”, que contiene los archivos de configuración del juego, así como un par de directorios más en los que se encontrarán a su vez los archivos de configuración del juego, pero en este caso ya personalizados y adaptados a los usuarios.

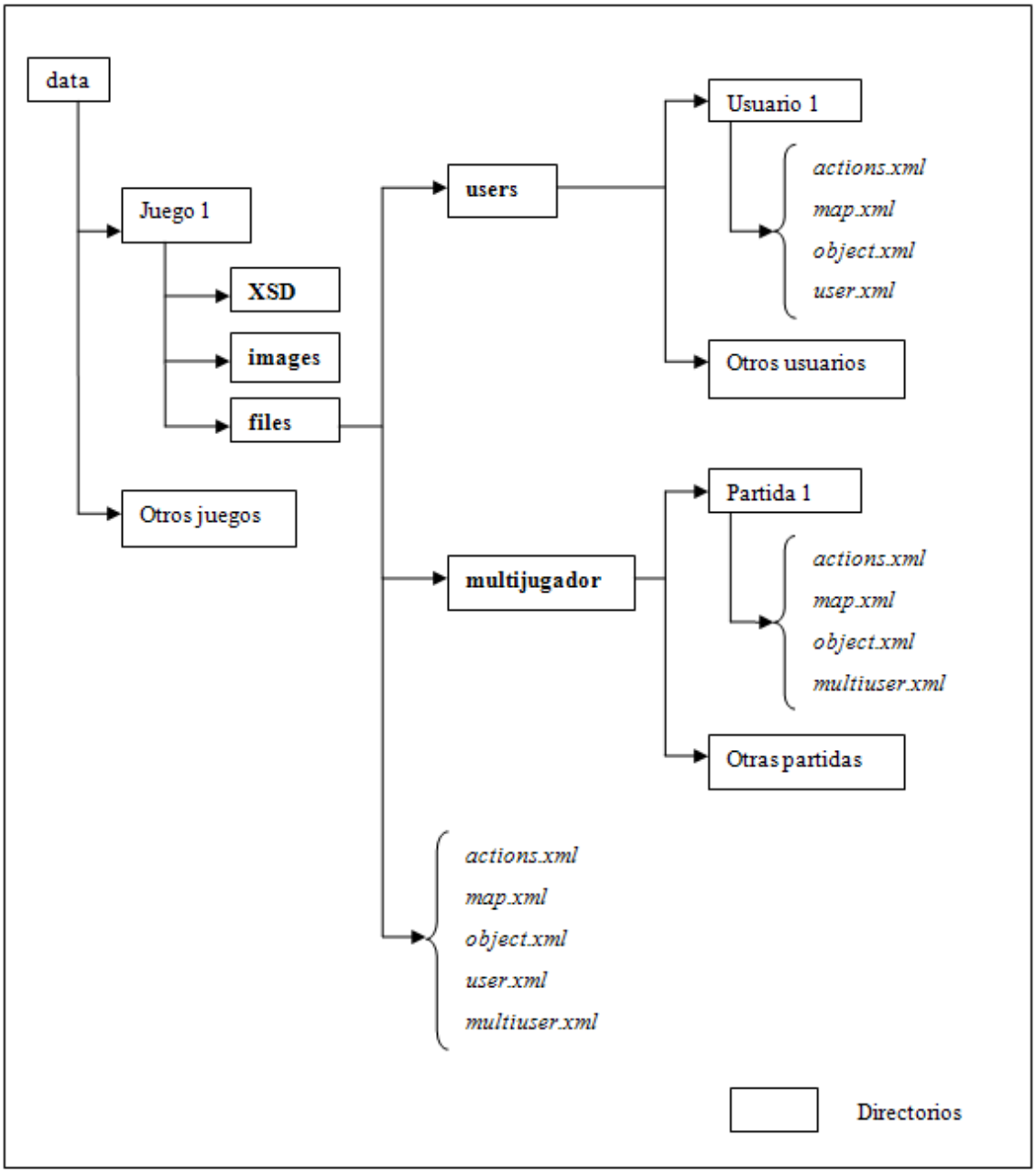


Figura 3.3 Estructura de archivos

3.4.1.2 Archivos de configuración del videojuego

Dentro del directorio "files", se encuentran cinco archivos de configuración del juego, entre ellos los cuatro archivos que poseía el motor original (*actions.xml*, *map.xml*, *object.xml*, *user.xml*), más un quinto archivo llamado *multiuser.xml*, en el que se guardan los progresos de los jugadores en una partida multijugador. Estos archivos no van a ser modificados en ningún momento por el motor, sino que, cuando se registre un nuevo jugador individual o una nueva partida multijugador, el motor usará estos archivos sin modificar para copiarlos al directorio del jugador o partida que corresponda, desde donde ya sí serán modificados y personalizados para reflejar el estado actual del juego. Además

de estos archivos, en el directorio “files” también se pueden encontrar dos directorios más, “users” y “multijugador”, dentro de cada uno de los cuales se puede encontrar una copia local de los archivos de configuración ya comentados, con las modificaciones pertinentes para que reflejen el estado del juego.

A continuación se detallarán los cambios que se hicieron al sistema de archivos, así como las nuevas funciones incorporadas a los videojuegos conversacionales desarrollados para la aplicación.

A. multiuser.xml

Del directorio “multijugador”, cuelgan una serie de directorios que corresponden a las distintas partidas con posibilidad de más de un jugador que se han creado. Dentro del directorio correspondiente al nombre de una de estas partidas, se pueden encontrar los tres archivos generales de configuración (*actions.xml*, *map.xml*, *object.xml*), como ocurre en el caso del directorio “users”, así como una copia del archivo *multiuser.xml*, que está modelado en la estructura del archivo *user.xml*, y dentro del cual se definen los parámetros de la partida multijugador. Se puede ver el contenido de un archivo *multiuser.xml* de ejemplo a continuación:

```
<config>
        <password>secreto</password>
        <player user="jugador1">
                <screen>1</screen>
        </player>
        <player user="jugador2">
                <screen>19</screen>
        </player>
</config>
```

Este archivo posee un elemento raíz llamado “<config>”, del cual cuelgan varios elementos hijos, “<password>”, y una serie de elementos “<player>”, uno por cada jugador que se haya inscrito a la partida. Así, una partida multijugador tiene un nombre único, que hace de identificador unívoco de la misma de cara al resto de jugadores, así como una contraseña global en este caso, que una vez introducida, da acceso al usuario que lo desee a entrar a formar parte de la partida. Esto es así porque es necesario permitir a los usuarios que así lo deseen la posibilidad de confirmar que sólo van a jugar con otros usuarios que conocen y a los que les han confiado la contraseña de la partida en la que juegan.

Por tanto, la etiqueta “<password>” contiene la contraseña general de la partida, y las etiquetas “<player>” contienen a cada uno de los jugadores inscritos en esta partida. Ésta última tiene un atributo “user”, en el cual se indica el nombre de usuario, y a su vez la etiqueta tiene un hijo llamado “<screen>”, cuyo valor es un identificador de la pantalla o escenario en el que se encuentra dicho jugador.

Conforme más usuarios se van sumando a la partida, se van añadiendo, uno a uno, más elementos “<player>”, para darles cabida. Es de notar que los nombres de usuario son locales a la partida, es decir, pueden repetirse a lo largo de distintas partidas sin indicar esto que, tras ellos, se encuentre el mismo jugador. Es una forma de hacer que el abanico de posibles nombres de usuario a elegir sea lo más amplio posible, y que siempre haya disponibles una gran variedad de nombres de usuario cortos y comunes, ya que se ha de tener en cuenta la incomodidad que supone el escribir a través de dispositivos que no tienen periféricos específicamente habilitados para ello, como es el caso de los teléfonos móviles.

B. actions.xml

En este archivo se detallan las acciones que puede llevar a cabo el usuario, así como las consecuencias que, en su caso, producen estas acciones sobre el entorno o los propios usuarios.

En los juegos desarrollados para el motor original, se nos presentaba un paisaje un tanto yermo. La causa principal era la ausencia de NPC que interviniesen en la historia, y con los que el jugador pudiese: conversar, extraerles información o incluso resolver acertijos para que le proporcionasen algún objeto necesario para seguir avanzando en la aventura.

Es por ello que se consideró oportuno añadir la existencia de NPC con los que se pudiera conversar, por medio de la opción “hablar con”. Cuando un jugador se encuentra con un NPC, puede decidir hablarle usando dicha opción, hecho lo cual se le presentan una serie de posibles frases que decirle. El jugador selecciona una de ellas, y la envía, a lo que el NPC contestará con otra frase. De ese modo, se establece una conversación, por la que el usuario puede ir progresando, hasta llegar a un momento en que no se le pueda extraer más información al NPC y por ello se vuelva al comienzo de la conversación, o a que se llegue al final del diálogo y se produzcan unas consecuencias determinadas, como por ejemplo la entrega de un objeto por parte del NPC al jugador. El guión de la conversación con el NPC

puede ser tan rico y extenso como se quiera, y puede ayudar a aumentar la duración de la experiencia de juego, si se van dando pistas en la conversación que hagan que el jugador explore a fondo las posibilidades de diálogo con los NPC que se vaya encontrando.

La etiqueta “<hablarcon>” indica que la acción de hablar con un NPC está disponible. Asimismo, el atributo “npc” indica el nombre del NPC al que se refiere este guión de conversación. En estos guiones, la conversación se estructura en niveles, denotados por el atributo “nivid” de la etiqueta “<nivel>”. Un nivel es un estado determinado de la conversación, que tiene asociado una serie de posibles frases que le puede decir el jugador al NPC, cada una de las cuales lleva una respuesta del NPC asociada.

```
<hablarcon npc="viejo">
        <nivel nivid="1">
                <lin linid="1" tonivel="1">¿Qué tal, abuelo?</lin>
                <resplin rlinid="1">He estado mejor. Hic.</resplin>
                <lin linid="2" tonivel="2">Buen día para echar un trago.</lin>
                <resplin rlinid="2">¡Siempre lo es!</resplin>
                <lin linid="3" tonivel="1">¡Qué buen día hace!</lin>
                <resplin rlinid="3">Si tú lo dices...</resplin>
        </nivel>
        <nivel nivid="2">
                <lin linid="1" tonivel="1">¿No le duele el lomo de estar ahí
                sentado?</lin>
                <resplin rlinid="1">Brmhbrm (murmullos ininteligibles)</resplin>
                <lin linid="2" tonivel="1">Quizás debería buscar un trabajo.</lin>
                <resplin rlinid="2">¡No hay trabajo en estas tierras para gente con mis
                habilidades!</resplin>
                <lin linid="3" tonivel="end">Me gusta su mondadientes. Le da un
                aire pensativo.</lin>
                <resplin rlinid="3">¡Ha ha ha! ¿Te gusta? Tuyo es.</resplin>
        </nivel>
        <nivel nivid="end">
                <text>El viejo te entrega su pegajoso y venerable mondadientes, tras
                lo que le vienen unas intensas ganas de dormir y se queda sopa.</text>
                <reacMod obj="viejo" atrib="screen" value="0"/>
                <reacMod obj="Viejo" atrib="screen" value="4"/>
                <reacMod obj="mondadientes" atrib="screen" value="user"/>
        </nivel>
</hablarcon>
```

El atributo “nivid” puede ser un número, en cuyo caso indica que sigue habiendo posibilidades de continuar la conversación entre el usuario y el NPC, o bien el valor “end”, lo cual indica que la conversación ha alcanzado su fin y que se va a dar paso a una serie de

consecuencias, como por ejemplo la entrega de un objeto por parte del NPC al jugador. Dentro de un determinado nivel que no sea el último, se encuentran las posibles frases que se le presentan al usuario para que elija cuál decir, recogidas bajo la etiqueta “<lin>”, así como la respuesta correspondiente a cada una de esas frases, en la etiqueta “<resplin>”. Además de sendos identificadores, en el caso de “<lin>” también se incluye un atributo “tonivel”, que indica el nivel de la conversación al que se va a progresar cuando dicha frase sea pronunciada por el protagonista de la historia. Como se ve en el ejemplo, se puede avanzar a un nivel posterior, retroceder a uno anterior, o ir directamente a “end”. Es de notar que el usuario puede abandonar la conversación en cualquier instante que lo desee, haciendo uso para ello del botón de despedida, que estará disponible durante toda la conversación. El atributo “end” denota un fin de la conversación, pero no uno cualquiera, sino el haber llegado a la situación en que la conversación acaba y se producen una serie de consecuencias, ya sea sobre el escenario, uno o más objetos, o incluso el propio jugador.

En el caso de que se encuentre el nivel “end”, es decir, de haber llegado al fin de la conversación, se indica en “<text>” el mensaje que se le va a mostrar al usuario en el móvil para informarle de que la conversación ha acabado, y a continuación se tendrán una serie de consecuencias, que en el ejemplo son de tipo “<reacMod>”, es decir, modifican a un objeto. En este caso, se modifica el objeto “viejo”, que es el NPC con el que se llevó a cabo la conversación; y se le asigna a su atributo “screen” el valor de 0, es decir, se elimina del mapa del juego. Además, se modifica el objeto “Viejo”, que es un personaje que ya no conversa con el usuario; y se sitúa en la pantalla en la que está el jugador. De este modo, cuando la conversación llega a su fin, se cambia a un NPC con una conversación definida por otro que no tiene nada que decir. Por último, se añade el objeto “mondadientes” al inventario del usuario.

Estas consecuencias no se limitan únicamente a la modificación de objetos, también se pueden modificar direcciones de movimiento posibles en un escenario, de forma que se abra un nuevo camino por el que ir (con “<reacModdir>”); o directamente transportar al usuario a otro escenario (con “<reacIra>”), por ejemplo si el personaje se desmaya y aparece más tarde en otra localización. También es posible desencadenar consecuencias que cambien la localización actual por otra distinta (con “<reacCambiaAlt>”), por ejemplo en el caso de que al acabar la conversación con un NPC, éste abandone el lugar, caso en el que habría que cambiar la imagen de la pantalla con el NPC presente por otra en la que ya no apareciese.

Por otra parte, una vez añadida la presencia de NPC, se vio como un paso natural el añadir una opción para “dar objeto a”, de modo que se pudiera entregar un objeto del inventario del jugador a un NPC.

```
<dara obj="botella" npc="viejo">
        <text>¡Ohh gracias! Que Dios te lo pague por obsequiarme con este brebaje
        de los cielos que... un momento... ¿vacía? ¡Ven aquí que te voy a dar yo! VEN
        AQUIIII!</text>
        <reacIra desp="Dead"/>
</dara>
```

En este ejemplo se puede ver que la etiqueta “<dara>” recoge la funcionalidad necesaria para llevar a cabo la acción. En su atributo “obj” se identifica el objeto que va a formar parte de la opción “dar objeto a”, y en el atributo “npc”, se identifica al NPC que va a ser el receptor de este objeto. Como se vio anteriormente, en <text> se recoge el texto informativo que se le va a presentar al usuario cuando se realice esta acción, y a continuación se presentan una serie de consecuencias que se producen, en este caso se envía al jugador a la pantalla correspondiente a su muerte.

Otra de las rutinas típicas de videojuegos de aventura conversacional es la existencia de un límite de tiempo en algunas localizaciones. Por ejemplo para notificar al usuario de que está pasando demasiado tiempo en la misma localización, o bien porque el lugar no sea seguro y pueda ocurrir algún evento si permanece allí un determinado tiempo. Este límite de tiempo se puede implementar de dos formas distintas. La primera, más rudimentaria, por medio de un sistema de turnos. Así, al entrar un jugador a una localización con límite de turnos, habría definido un número máximo de acciones que puede realizar ese usuario en la localización. Al agotar ese número de turnos, se daría paso a determinadas consecuencias. La segunda forma, más elaborada, consiste en establecer unos límites de tiempo dentro de la localización. Por lo que, por ejemplo, se puede indicar al sistema que si el usuario lleva más de 10 segundos en una determinada localización, ocurran unas consecuencias.

En este proyecto se ha elegido la segunda forma. Para implementarla, en el archivo *map.xml*, se indica en todas las localizaciones del juego si tienen definido un tiempo límite o no. Para las localizaciones que tengan definido dicho tiempo límite, será necesario conocer cuál es. Para ello, se accede al archivo *actions.xml*, y se busca la acción de tipo

<timedAction scr=”x”>, donde el atributo “scr” tiene que coincidir con la localización que tiene definido el tiempo límite.

```
<timedAction scr="7">
        <timeAt tid="1" next="end" t="15">El suelo empieza a resquebrajarse...
        mejor harías en salir de aquí por piernas.</timeAt>
        <timeAt tid="end" t="10">
                <text>Tu peso, junto con la imprudencia de ponerte a deshojar
                margaritas en el borde de un precipicio, hacen que el suelo se hunda y
                tengas tickets en primera fila para ver una caída de 500 metros de
                altura.</text>
                <reacIra desp="Dead"/>
        </timeAt>
</timedAction>
```

Dentro de una etiqueta “<timedAction>” cuelgan una o más etiquetas “<timeAt>”, cada una de las cuales define un temporizador para esa localización, locales a cada usuario. Cuando un usuario entra por primera vez a la localización con temporizador, se accede a la primera etiqueta “<timeAt>”, identificada por tener su atributo “tid” igual a 1. En el atributo “t” se indica la duración de ese temporizador en segundos. Y se define un texto a mostrar al usuario cuando éste expire.

A continuación, en el atributo “next”, se define cuál es el atributo “tid” que identifica al siguiente temporizador. En este caso es “end”, lo que indica que será el último y provocará unas determinadas consecuencias al agotarse. Al ser el último temporizador de la localización, es distinto a los demás. No contiene un atributo “next” al no tener que definir el siguiente temporizador. De nuevo en “t” se indica la duración del temporizador que, como se puede ver, no es acumulativa. Es decir, este segundo temporizador tiene una duración de 10 segundos, que comenzarán a contar si se deja expirar el primer temporizador de 15 segundos. Por lo que, desde que el usuario entra a esa localización hasta que ocurren las consecuencias pasan 25 segundos en total

Asimismo, el último temporizador incluye bajo la etiqueta “<text>” un texto a mostrar al usuario en forma de alerta tras ejecutarse las consecuencias que ocurren al agotarse el tiempo. Por último, se incluyen varias consecuencias que pueden ser de cualquier tipo, en este caso una consecuencia de tipo “<reacIra>” que causa la muerte del jugador.

Todos los temporizadores de una localización se reinician cuando el usuario la abandona, de modo que si el usuario del ejemplo ha visto el primer aviso y se mueve a otro lugar, cuando vuelva a entrar a la localización con tiempo límite tendrá de nuevo a su disposición todo el tiempo que ofrecen los temporizadores definidos, y no una fracción de éste.

Siguiendo con los elementos añadidos a *actions.xml*, también fue necesario añadir un nuevo tipo de consecuencia a las ya existentes, denotada por “<reacCambiaAlt>”, para tener cubierto el caso de que haya cambios sustanciales en una localización (por ejemplo, que cambie su descripción y/o imagen que la ilustra). En ese caso, con una orden de tipo “<reacCambiaAlt>”, e indicando con su atributo “escen” el identificador de la “<screen>” o localización de *map.xml* que va a cambiar por su versión alternativa, se consigue cambiar para todos los jugadores de la partida la localización por una versión modificada de la misma. Hay que tener en cuenta que las consecuencias de tipo “<reacCambiaAlt>” son condicionales, es decir, no siempre se ejecutan.

```
<usarcon obj1="Linterna" obj2="oscuridad">
        <text>Alumbras a tu alrededor.</text>
        <reacMod obj="charco" atrib="screen" value="3"/>
        <reacCambiaAlt useren="3" escen="3" nuevavers="iluminada"/>
        <!--Cambiamos ésta por la pantalla del pozo con luz -->
</usarcon>
```

Para que una consecuencia de este tipo se ejecute, se tiene que cumplir que su atributo “useren” coincida con la localización del usuario que genera la consecuencia (en el ejemplo, para que solo se ejecute el cambio por la versión alternativa si se usa la linterna encendida con la oscuridad estando en la localización correspondiente al pozo). Además, se tiene que cumplir que el valor del atributo “nuevavers” sea distinto a la versión actual de la localización (en el ejemplo, si se tiene la localización del pozo en su versión iluminada, no es necesario cambiarla por su alternativa, sin embargo sí que habría que hacerlo si se tuviese la versión oscura de la pantalla). Como se puede ver, es posible realizar el cambio de una localización por su versión alternativa (señalada por “escen”) sin ser necesario que el usuario esté presente en ella, aunque sí tendrá que estar en el lugar oportuno para generar las consecuencias que provocan el cambio (denotado por “useren”).

Esto se puede ver con más detalle bajo el epígrafe que discute los cambios en *map.xml* más adelante.

Tras añadir las consecuencias del tipo “<reacCambiaAlt>” se realizó un análisis de las situaciones que se podrían implementar en un videojuego en las que sería útil, y se vio necesario incluir dos elementos adicionales en *actions.xml* para completar la funcionalidad.

En primer lugar, es posible que el cambio que se produce de una localización por su versión alternativa esté ligado a un objeto que se usó para provocar ese cambio. En el ejemplo visto, es la linterna encendida la que provoca el cambio de la versión oscura de la localización del pozo a la iluminada. Por ello, es necesario definir una acción que se ejecute cuando el usuario que tiene el objeto que provocó y mantiene el cambio abandone la localización, para que ésta pueda volver a su estado original. De este modo, al abandonar la pantalla del pozo iluminado el usuario que lleva la linterna encendida, se pasaría de nuevo a la versión del pozo oscuro. Esto se regula con la etiqueta “<iraalt>”.

```
<iraalt fromscr="3" obj="Linterna">
        <reacCambiaAlt useren="3" escen="3" nuevavers="oscura"/>
        <!--volvemos a la versión oscura de la pantalla -->
</iraalt>
```

En el atributo “fromscr” se define la localización a cambiar que tiene que estar abandonando el usuario que porta el objeto ligado al cambio, denotado por el atributo “obj”. Así, cuando el usuario abandone el pozo con la linterna encendida, si el pozo está iluminado, se cambiará por su versión oscura.

En segundo lugar, se vio que podía ser necesario crear una diferenciación que permitiese distinguir entre los objetos que se dejan en una y otra versión de una localización. En el ejemplo, si un usuario quisiera dejar un objeto en la pantalla del pozo, lo lógico es que ese objeto sólo apareciese en su versión iluminada, ya que en la versión oscura no se puede ver nada. Por ello, se define una nueva etiqueta de tipo “dejaralt”.

```
<dejaralt scr="3" vers="iluminada"/>
```

De este modo, se indica que cualquier objeto que se quiera dejar en la pantalla que indica el atributo “scr” (que corresponde en el ejemplo al pozo), se va a dejar en la versión que indica “vers” de esa pantalla (en el ejemplo, la versión iluminada de esa localización). Esto se verá en más detalle en el epígrafe correspondiente a modificaciones en *object.xml*.

C. object.xml

En este archivo se detallan todos los objetos existentes en el mapa, así como el lugar en el que se encuentran, si se pueden utilizar, si se pueden coger, etc. Al haber añadido la posibilidad de que ciertas localizaciones tengan una versión alternativa definida, se añadió un atributo “vers” que indica la versión de la localización (si la hay) en la que el objeto en cuestión está visible. Si un objeto tiene definido este atributo, al cogerlo se eliminará el valor de “vers”, ya que sólo tiene sentido para objetos que estén en el escenario, no en el inventario de un jugador.

```
<ob id="moneda" screen="3" vers="iluminada" cogible="si" usable="no" npc="no">
```

Asimismo, al incorporar la opción de NPC, fue necesario también añadir un atributo “npc” en la etiqueta “<obj>”, de modo que se indique si el objeto en cuestión es un NPC o no.

```
<ob id="pozo" screen="2" vers="" cogible="no" usable="no" npc="no">
```

En el caso de un NPC, será necesario, además de identificarlo como tal asignándole “si” al atributo “npc”, añadir una nueva etiqueta “<user>”.

```
<ob id="viejo" screen="4" vers="" cogible="no" usable="no" npc="si"> Puedes ver a
un extraño viejo en la ribera del río.
        <mirarno>Es un viejo harapiento, parece ligeramente achispado.</mirarno>
        <errordara>Creo que solo le interesa lo que pueda haber en el fondo de una
        botella</errordara>
        <user usid="" nivel=""/>
        <user usid="jugador1" nivel="2"/>
</ob>
```

En esta etiqueta, por medio del atributo “usid”, se indica que se ha iniciado una conversación con un usuario, en el ejemplo, “jugador1”. La primera etiqueta “<user>”, que aparece con sus atributos vacíos, no cambia nunca; y se usa por parte del motor para añadirla al final de la lista de usuarios e insertar el nombre de usuario y el nivel por el que va la conversación cada vez que un nuevo usuario le hable al NPC. De ese modo, se lleva un control de por dónde va la conversación de cada usuario con el NPC, en el caso de que se tenga una partida multijugador.

Asimismo, se definen también un par de nuevos campos de error con las etiquetas “<errorhablar>” y “<errordara>”, que se usan para informar al usuario cuando no es posible realizar estas acciones con el objeto seleccionado. Estos campos se definen al comienzo del archivo de forma general, y se pueden redefinir en cada uno de los objetos para dar lugar a un tratamiento más diferenciado, como se puede ver en el ejemplo anterior en el caso de que al NPC se le ofrezca otro objeto que no sea una botella.

D. map.xml

En este archivo es donde se definen los aspectos más estáticos del juego, desde los textos que aparecen en cualquier formulario o botón de la aplicación, hasta las localizaciones del juego, junto con las direcciones a las que se puede desplazar el jugador, los textos descriptivos y las imágenes que los acompañan.

En este proyecto, y dado que se añadió entre otras cosas el modo de juego multijugador, así como varias acciones más como “hablar” con un NPC o con otro usuario, y “dar objeto a” un NPC u otro usuario; fue necesario incluir diversos nuevos textos en los archivos de configuración para dar cabida a las nuevas posibilidades de la plataforma, para lo que se utilizaron etiquetas adicionales bajo los epígrafes correspondientes de *map.xml*. Por ejemplo, bajo “<screenIndex>” se definen los textos e imágenes que aparecen en la presentación de la aplicación, por lo que fue necesario añadir en su interior las etiquetas “<singleCom>” y “<multiCom>”, en las que se muestra el texto que se le presenta al usuario para elegir el modo de un jugador o multijugador, respectivamente.

Lo mismo ocurre en el epígrafe “<screenLogin>”, donde se definen los textos que van a ser mostrados al usuario cuando esté registrando un nuevo nombre de usuario, creando una nueva partida, entrando en una ya existente o recuperando el progreso que tenía guardado en el juego bajo su nombre de usuario. En el ejemplo a continuación se muestran algunos textos añadidos a este epígrafe.

```
<screenLogin>
        <title>Regístrate</title>
        <!-- Titulo de esta pantalla -->
        <textUser>Usuario: </textUser>
        <!--Texto para pedir el nombre del usuario en el formulario -->
        <textLoginSingle>Entrada un jugador</textLoginSingle>
        <!--Texto para seleccionar entrada para un jugador -->
        <textLoginMulti>Entrada Multijugador</textLoginMulti>
        <!--Texto para seleccionar entrada a partida multijugador -->
</screenLogin>
```

También fue necesario incorporar a la lista de posibles acciones a realizar por el usuario, las nuevas acciones definidas, en este caso “hablar con” y “dar objeto a”, que se ubican bajo el epígrafe “<screenShowInfo>”, donde se recogen los textos a mostrar en la pantalla principal de juego.

Para dar cabida a los nuevos límites de tiempo definidos en ciertas localizaciones, fue necesario modificar la forma de las etiquetas “<screen>” añadiéndoles el atributo “timed”, que puede tomar el valor sí o no. En este atributo se indica si la localización tiene definido algún temporizador o no.

Por último, también hubo que realizar modificaciones en el diseño del archivo *map.xml* para dar cabida a la funcionalidad multijugador. Por ejemplo, en el caso de que en un escenario se realicen cambios esenciales, que varíen el texto descriptivo de la situación así como la imagen que ilustra la localización; en el motor antiguo se optó por utilizar una nueva localización y transportar allí al jugador, creando para ello una nueva entrada “<screen>” en este archivo. Este caso se puede dar, por ejemplo, si el usuario se halla en una habitación oscura, encuentra un interruptor y lo acciona, hecho que causa que se encienda la luz y por tanto cambie tanto la imagen que ilustraba el escenario como la descripción del mismo. Sin embargo, en el caso de una partida multijugador, este sistema no es directamente extensible, ya que no hay una forma sencilla de que se transporte también a los jugadores que pudieran estar compartiendo la pantalla con el que acciona el interruptor. Por lo que este último sería transportado correctamente a la pantalla iluminada, mientras deja a sus compañeros en la oscuridad.

Para solucionar esta eventualidad, se decidió incluir un nuevo epígrafe denotado por la etiqueta “<alt>” bajo el elemento “<screen>” que recoge todo lo necesario para mostrar un escenario alternativo cuando sea preciso. Además, se añadió un nuevo atributo llamado

“version” a todos los elementos “<screen>”, en el que se indica la versión actual de la localización si es una localización con versión alternativa. Si no, se deja vacío.

```
<screen id="3" version="oscura" timed="no">
        <title>Pozo</title>
        <text>Este sitio apesta, y está muy oscuro. </text>
        <img>/AO/images/ORIGINAL_3.png</img>
        <direction>
                <d id="2" activ="si">Subir</d>
        </direction>
        <alt>
                <title>Pozo</title>
                <text>Puaf, El fondo del pozo está casi seco (por suerte para ti,
                curiosón). Solo quedan algunos charcos malolientes de agua
                estancada.
                La estrechez de las mohosas paredes y el enorme silencio que reina
                hacen que tengas por única compañía el latido desbocado de tu
                corazón.</text>
                <img>/AO/images/ORIGINAL_26.png</img>
        </alt>
</screen>
```

De esta forma, se define una localización alternativa bajo la etiqueta “<alt>”, que recoge los campos “<title>”, “<text>” e “<img>” de un escenario “<screen>” normal, y en el caso de que ocurra alguna acción que tenga como consecuencia la orden de cambiar el escenario por su versión alternativa, se hace un cambio entre los dos, y se actualiza a todos los jugadores que allí se encuentren, por lo que todos ellos pasarían a estar en la habitación iluminada, en el ejemplo propuesto.

Hay que reseñar que, al comienzo de los archivos de configuración XML en el videojuego conversacional desarrollado a modo de prueba, “La Aventura Original”, se detallan en forma de comentarios todas las modificaciones realizadas al motor original, así como la función que desempeñan los nuevos añadidos.

3.4.2 CLIENTE

3.4.2.1 Introducción

La aplicación desarrollada en el presente proyecto está compuesta por tres partes fundamentales, una que reside en el servidor remoto, y que se encargará de proporcionar los contenidos a los dispositivos móviles, así como de llevar la carga de la inteligencia del sistema. Otra, que reside en los terminales móviles que utilizarán los usuarios para acceder

al servicio de videojuegos conversacionales. Y por último, la tercera parte, que se refiere al protocolo de comunicación desarrollado para posibilitar un entendimiento entre las dos anteriores, clientes y servidor.

Este apartado está centrado en la parte referida al cliente móvil de la aplicación. Este cliente debe proporcionar la funcionalidad de la vista de la aplicación, por lo que proveerá el interfaz gráfico que se presente a los jugadores. Para ello, será necesario tener un *MIDlet*, que permita mostrar a los jugadores las localizaciones del juego, sus descripciones e imágenes que las acompañan, y las acciones a realizar. Este último requisito hace que el *MIDlet* tenga que estar preparado para recibir las órdenes del usuario, convertir esas órdenes en acciones que el servidor entienda, y pasarlas a un módulo de comunicaciones para que se encargue de enviarlas y continuar con el flujo de juego.

Además del *MIDlet*, y del módulo de comunicaciones mencionado, se desarrollaron otras clases auxiliares. En la Figura 3.4 se puede ver la relación de todas estas clases.

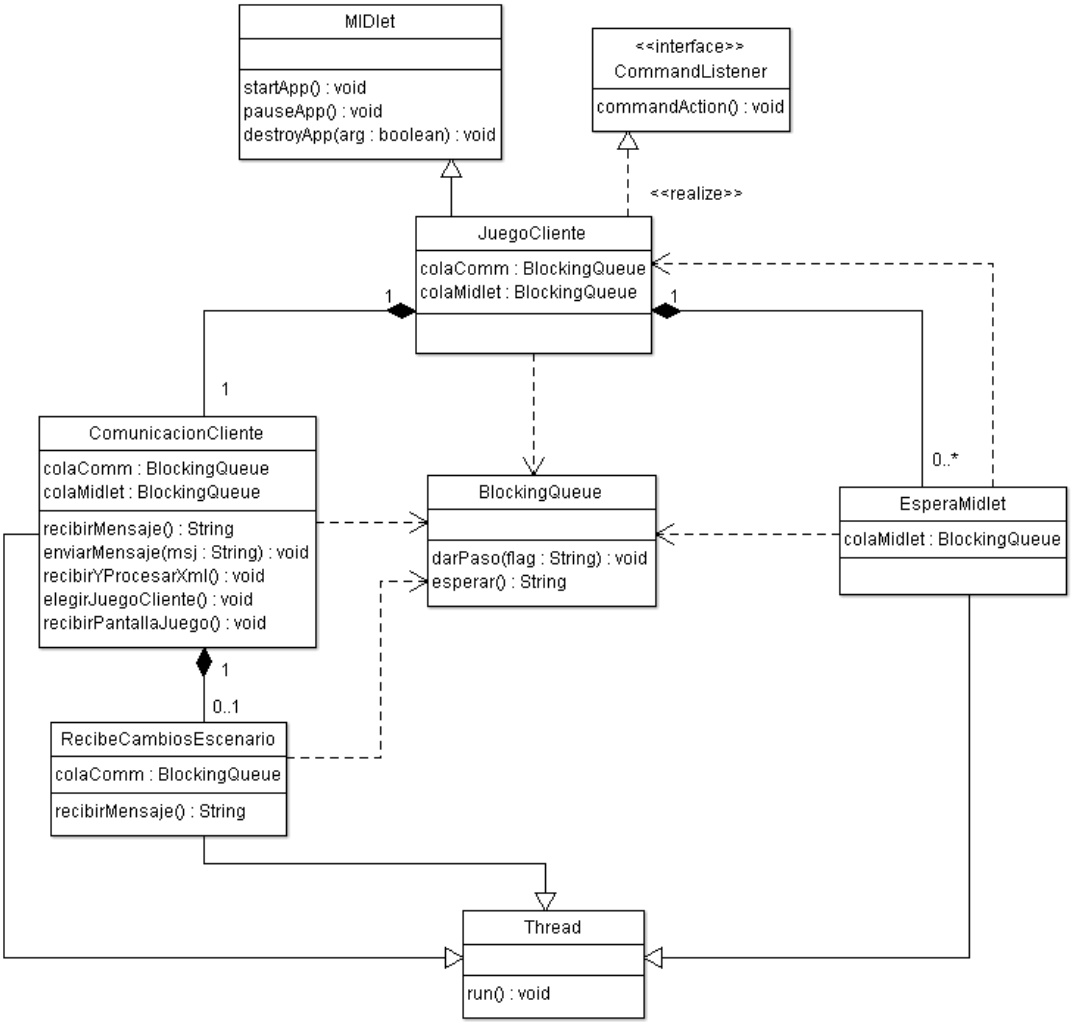


Figura 3.4 Diagrama de clases del cliente

Como se puede ver en el diagrama de clases del cliente, la clase principal es *JuegoCliente*, que, como se ha comentado, extiende a *MIDlet*, y por ello ha de implementar los métodos *startApp()*, *pauseApp()* y *destroyApp()*. Asimismo, como va a encargarse también de interaccionar con el usuario, implementa el interfaz *CommandListener*, que proporciona un modo de realizar la escucha de eventos del usuario, que serán atendidos por el *MIDlet* de la forma que corresponda. Estos eventos van a ser atendidos en el método *commandAction ()*, al que se llama cada vez que el usuario interaccione con la aplicación y seleccione un comando.

3.4.2.2 Sistema de turnos

Por debajo de *JuegoCliente*, se puede ver la clase *BlockingQueue*. Esta clase implementa una cola síncrona o cola de bloqueo, que es una cola de tamaño máximo igual a uno, por lo que en cualquier momento sólo puede estar vacía (si no contiene ningún elemento) o llena (si contiene un elemento). Además de esto, la característica principal de una cola síncrona viene indicada en su propio nombre, y es que tanto el añadir un elemento como el retirarlo de la cola, se hacen de forma síncrona. Es decir, si se intenta agregar un elemento a la cola, y ésta está vacía, se hace inmediatamente. En cambio, si la cola está llena en el momento de agregarle dicho elemento, el hilo de ejecución se detiene y espera hasta que otro hilo haya sacado un elemento de ella, para poder añadir el suyo. De igual manera, si se intenta extraer un elemento de la cola cuando ésta está llena, la acción se lleva a cabo en el momento. Si, por el contrario, está vacía, el hilo de ejecución se detendrá y esperará hasta que otro hilo introduzca un elemento en la cola que pueda ser extraído. Este comportamiento característico hace que las colas síncronas sean muy utilizadas para regular accesos a modo de testigo o semáforo en entornos de ejecución multihilo.

En la clase *BlockingQueue*, se dispone de dos métodos de acceso a la cola: *darPaso(String flag)*, que introduce el elemento *flag* en la cola y, por tanto, da paso a un hilo que estuviese esperando para extraerlo; y *esperar()*, que extrae un elemento de la cola, y en el caso de que no esté disponible inmediatamente, esperará hasta que otro hilo lo introduzca para extraerlo.

En el cliente va a ser necesario establecer un mecanismo de turnos. Por una parte, cuando el usuario seleccione una acción en el juego y pulse el botón de enviar en su móvil, habrá que mostrar una pantalla de espera sin posibilidades de interacción (por ejemplo, que muestre una imagen de un reloj de arena y el mensaje “realizando operaciones de red”,

para que el usuario sepa que, aunque no se le permite hacer nada, es un comportamiento previsto y que ha de llevar relativamente poco tiempo), para que no se intente hacer otro envío por la red mientras el servidor no ha llegado a contestar aún al primero. Para esto se usa una cola síncrona *colaMidlet*, que, por medio de su método *colaMidlet.esperar()*, hará que el *MIDlet* espere en esa pantalla sin posibilidades de interacción hasta que llegue la contestación del servidor al módulo de comunicaciones y desde éste se llame a *colaMidlet.darPaso(flag)*, indicando que, como ya se ha recibido una respuesta por parte del servidor, se puede acceder a las variables que se han recibido de éste. La variable *flag* es un texto que se introduce en la cola síncrona a modo de testigo, y que no tiene ninguna utilidad real (se podría introducir cualquier objeto en la cola), más que la de poder trazar de dónde viene ese testigo si en el momento de introducirlo, se le asignó un valor que lo explique.

Sin embargo, existe un problema al usar esta aproximación. Como se ha dicho, el *MIDlet* se encarga de llevar la interacción con el usuario por medio de la escucha de eventos, que se producen cuando el usuario selecciona algún comando. De ese modo, el *MIDlet* es el hilo de ejecución principal de la aplicación, por lo que no se puede poner en espera, ya que, de hacerlo, no habría forma de despertarlo más adelante, causando un error en la aplicación, que quedaría bloqueada. Es por ello que, para realizar esta pausa, se usó una clase auxiliar *EsperaMidlet*, que el *MIDlet* lanza justo tras mostrar la pantalla de espera al usuario, y cuyo único cometido es realizar la espera con *colaMidlet.esperar()* hasta que el módulo de comunicaciones le indique que se ha terminado de realizar el diálogo en red con el servidor con *colaMidlet.darPaso(flag)*, momento en que *EsperaMidlet* llama a uno de los métodos de *JuegoCliente* que se encargan de mostrar, de nuevo, una pantalla de juego o uno de los distintos formularios en los que el usuario selecciona el juego al que quiere jugar, su información de entrada en forma de nombre de usuario y contraseña, etc.

Por otra parte, la segunda espera viene dada por el hecho de que el módulo de comunicaciones es un hilo, y tiene definido en su método *run()* un flujo que ha de seguir, en el cual va recibiendo y enviando mensajes. Sin embargo, una vez que ha recibido del servidor los contenidos necesarios para mostrar al usuario una localización del juego completa, es necesario hacer que este hilo espere hasta que el usuario haya decidido realizar una acción determinada. Así, por medio de la cola síncrona *colaComm* se implementa esta característica, de modo que cuando el módulo de comunicaciones recibe

los contenidos del servidor y los pone a disposición del cliente haciendo, por ejemplo, *colaMidlet.darPaso("Lista de juegos preparada")*, es necesario detener la ejecución de ese hilo hasta que el usuario decida qué juego elegir, lo que se hace con *colaComm.esperar()*. Más tarde, será el cliente el que, una vez elegida la acción a realizar, ejecute *colaComm.darPaso()*, indicando así al módulo de comunicaciones que puede seguir adelante y ya sí volver a comunicarse con el servidor, ya sabiendo la acción que ha elegido tomar el jugador.

Ya se han explicado al detalle las interacciones que tienen las clases *JuegoCliente*, *EsperaMidlet* y *ComunicacionCliente* con las colas síncronas que implementa *BlockingQueue*. Sin embargo, en el diagrama se ve que hay una cuarta clase que también tiene este tipo de relación de dependencia con las colas síncronas, que es *RecibeCambiosEscenario*.

RecibeCambiosEscenario, junto con *ComunicacionCliente*, forman el módulo de comunicaciones del cliente, y son las clases que proporcionan una comunicación con el servidor por medio de sockets a través de la red. *RecibeCambiosEscenario* establece una nueva conexión con el segundo puerto del servidor. Esto ocurre tras recibir la primera localización del juego, para poder proporcionar en todo momento una escucha del cliente al servidor para ver si hay peticiones de actualización asíncronas de éste. Estas peticiones pueden indicar varias cosas: que se estaba en una localización con tiempo límite y ha transcurrido este tiempo (el único tipo de peticiones de actualización asíncrona que van a ocurrir en una partida individual), que ha habido cambios en el modelo relevantes a la localización en la que está el usuario, o que ha habido cambios en el entorno de juego (por ejemplo, que otro jugador ha entrado o salido de la localización en la que se encuentra el usuario).

Si ha expirado un temporizador en la localización actual, si otro jugador ha causado cambios relevantes en el modelo, o si un jugador ha entrado o salido de la localización; el servidor avisa a *RecibeCambiosEscenario* mandándole varios mensajes en los que indica el tipo de procedimiento a seguir por el cliente. Por último, se llama a *colaComm.darPaso(flag)*, de modo que se saca de la espera a *ComunicacionCliente*, y en *flag* se le indica que se le ha despertado y el motivo. Tras esto, *ComunicacionCliente* procede a llamar al método de *JuegoCliente* que hace que se muestre la pantalla de espera, y pide la actualización al servidor. Al final de este diálogo, guarda los valores actualizados

del modelo que eran relevantes para el cliente, y permite continuar el juego. De este modo, *RecibeCambiosEscenario* se encarga de recibir aviso de los cambios asíncronos que se hayan producido en el contexto de juego y que sean relevantes al usuario (por ejemplo, cambios en la localización en la que se encuentra), y señalizar a *ComunicacionCliente* para que inicie un procedimiento para traer del servidor la versión más actualizada del modelo. Esto ocurre también en el caso de que haya salidas o entradas de otros usuarios en la localización en la que el jugador se encuentra, ya que en todo momento tiene que conocer cuáles son los jugadores que comparten escenario con él, debido a que hay acciones que los involucran, como por ejemplo "hablar con" o "dar objeto a".

3.4.2.3 Recepción de contenidos en el cliente

La clase encargada en el cliente de recibir los contenidos que proporciona el servidor es *ComunicacionCliente*. Esta clase utiliza un socket para establecer una comunicación bidireccional con el servidor, dejando que dicho socket permanezca abierto durante la duración de toda la sesión de juego.

En esta clase, además de tener métodos necesarios para recibir y enviar mensajes, que serán imprescindibles para llevar a cabo el protocolo de comunicación implementado y del que se hablará más adelante, existen métodos para recibir los contenidos que han de ser adaptados para ser mostrados por pantalla en el cliente. Estos contenidos van a ser de dos tipos: variables que contienen información proporcionada por el servidor como resultado a una petición acompañada de unos ciertos datos del cliente, y por otra parte los que forman el esqueleto de la aplicación (títulos de formularios, botones, campos de texto, etc.):

a) Contenidos dinámicos: en el primer tipo de contenidos que va a recibir el cliente, el que corresponde a respuestas del servidor a peticiones previas que han de ser procesadas, se enviarán los datos directamente por el socket que queda establecido entre cliente y servidor. Para ello se utilizarán métodos que agruparán la recepción de estos datos, por ejemplo *elegirJuegoCliente()*, que se encarga de recibir la información de la presentación del juego elegido por el usuario; o *recibirPantallaJuego()*, que recibe del servidor todos los datos necesarios para construir un escenario normal de juego (recibiendo las variables correspondientes a las direcciones en las que puede moverse el jugador, los objetos que hay en pantalla, los objetos disponibles en el inventario del jugador, etc.).

b) Contenidos estáticos: en cuanto al segundo tipo de contenidos, los contenidos estáticos que forman la estructura de los formularios y botones de la aplicación, se decidió realizar el envío usando el formato XML, por la sencillez y claridad que proporciona su uso, frente a tener que enviar una a una gran cantidad de variables individuales. Además, estos datos ya se encontraban en el servidor en formato XML, por lo que era aconsejable aprovecharlo. Sin embargo, esta decisión conlleva que en el cliente se reciban fragmentos de archivo XML (ya que sólo se envía la parte relevante al lugar del flujo de la aplicación en el que se encuentre el cliente, para hacer que las transferencias sean menos pesadas y el comportamiento en general de la aplicación sea más fluido), por lo que será necesario tener un *parser* XML en el cliente para poder extraer los contenidos.

En el capítulo 2 se comentó la elección del *parser* Xparse-J por sus especiales características de ser software libre, ser compatible con MIDP, ser ligero y ser de tipo *model*, lo que hace que cargue en memoria el contenido de los archivos XML y se pueda navegar por las etiquetas y atributos a modo de árbol, de forma sencilla e intuitiva. El método *recibirYProcesarFragmentoXml()* es el encargado de ir recibiendo los fragmentos del archivo *map.xml* relevantes a la situación en la que se encuentra el usuario, que se corresponderá con una de las que recogen los siguientes campos de este archivo:

- “<screenIndex>”, que incluye los textos de la pantalla de presentación del juego.
- “<screenLogin>”, que incluye los textos del proceso de registro de usuario individual o partida colectiva, y los de entrada a una partida ya creada, sea ésta individual o colectiva.
- “<screenRestart>”, que incluye los textos que se muestran en el evento de que se quiera reiniciar una partida.
- “<screenShowInfo>”, que incluye los datos necesarios para mostrar la pantalla principal de juego formada por una imagen de la localización y un texto descriptivo, así como una lista de posibles acciones a realizar por el usuario.

De estos fragmentos recibidos, se extraerán los textos descriptivos, textos de botones y comandos, y direcciones de imágenes en el servidor; necesarios para poder construir y mostrar la pantalla pertinente al usuario en cada momento.

3.4.2.4 Pérdida de conexión con el servidor

Una vez que se está en el flujo de juego siempre existe la posibilidad de que, aún habiéndose establecido correctamente la conexión inicial y haber comenzado a jugar, haya algún problema que haga que se pierda la conexión con el servidor. Esta eventualidad está prevista por medio del tratamiento de excepciones en el cliente. En el caso de que haya un problema con la conexión, se producirá una excepción de tipo *IOException* (excepción que se produce cuando hay un error en operaciones de entrada o salida) en *ComunicacionCliente* cuando se intente realizar un envío desde el cliente. Esa excepción se captura en el propio *ComunicacionCliente* y desde allí se utiliza un método de *JuegoCliente* para mostrar una alerta al jugador en la que se le indique que ha habido un problema con la conexión.

Tras aceptar el mensaje de información mostrado en la alerta, al usuario se le presenta una lista con tres opciones: intentar reconectar con el servidor, cambiar los datos de configuración del servidor y abandonar la aplicación. Al haberse producido el error tras haber estado un determinado tiempo jugando correctamente, los datos de configuración del servidor han de ser correctos. Si se elige la opción de intentar reconectar con el servidor, el cliente intentará establecer la comunicación con éste lanzando una nueva instancia de *ComunicacionCliente* desde el *MIDlet*. Si no logra conectar, volverá a mostrar la elección entre las tres opciones comentadas para que el usuario pueda seguir reintentando la conexión si lo desea, o abandonar la partida. En cambio, si se vuelve a establecer la conexión con el servidor se procederá a comenzar un nuevo diálogo con éste desde el principio. No se habrán perdido los progresos realizados en la partida ya que se guardan en los ficheros XML de configuración tras cada acción realizada por el usuario.

3.4.2.5 Vista de la aplicación

J2ME ofrece dos opciones a la hora de crear la interfaz de usuario de un *MIDlet*: una enfocada al diseño de la interfaz de usuario a bajo nivel, y otra a alto nivel. Cualquier elemento que se puede mostrar por pantalla en MIDP tiene que heredar de la clase *Displayable*. Esta clase tiene, a su vez, dos subclases, que son *Canvas* y *Screen*, como se puede ver en la Figura 3.5. *Canvas* forma la API de bajo nivel, teniendo métodos que permiten acceder directamente a la pantalla y “dibujar” sobre ella (de ahí su nombre, ya que en inglés “canvas” significa lienzo). Esto permite un amplio control de la apariencia que va a tener la pantalla en todo momento, si bien tiene como contrapartida la

portabilidad, ya que cuanto más control se ejerza sobre un elemento, menos portable a otros dispositivos acaba siendo.

Por su parte, *Screen* es la API de alto nivel, que proporciona una serie de elementos con un gran nivel de abstracción, por lo que no se tiene un control preciso de su representación en todo tipo de dispositivos, sin embargo sí que se tiene una gran portabilidad en todo el rango de dispositivos que soportan MIDP, así como la comodidad de no tener que construir desde cero los bloques fundamentales que se van a usar para crear la interfaz de usuario. Sin embargo, para poder utilizar esta API, es necesario que sea suficiente con los elementos que proporciona para crear la interfaz gráfica. Estos elementos son, entre otros:

- *Form*, que implementa un formulario genérico al que se le pueden insertar otros elementos, como imágenes, textos, cajas de texto editables, listas de opciones, etc.
- *List*, que crea una lista de opciones en pantalla. No se puede insertar en un *Form*.
- *TextBox*, que genera una caja de texto que el usuario puede utilizar para introducir texto en ella.
- *Alert*, que muestra por pantalla un aviso o alerta de que la acción realizada está fuera del flujo de control establecido, por ejemplo porque se ha dejado un campo en blanco en el formulario de registro.
- *StringItem*, que muestra un fragmento de texto en la pantalla.
- *ChoiceGroup*, que es una lista similar a la que proporciona *List*, pero con la diferencia de que se puede insertar dentro de un *Form*.

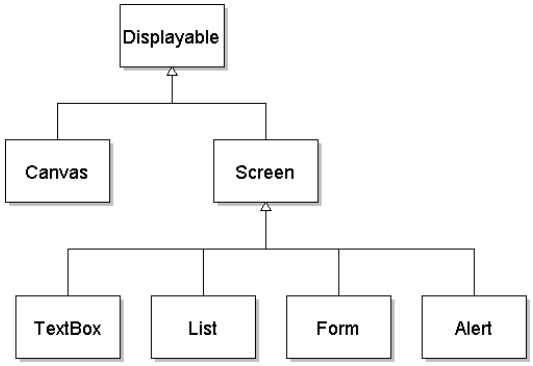


Figura 3.5 Clases que implementan la interfaz de usuario en MIDP

Para implementar la parte de la vista que propone la arquitectura MVC, se tuvo en cuenta un análisis del tipo de información que se tenía que mostrar en pantalla en un videojuego de aventuras conversacionales: una imagen a modo de ambientación de la localización, un texto descriptivo que informe de la situación y proporcione posibles pistas

sobre la acción a tomar, y por último una lista de posibles acciones que puede llevar a cabo el jugador. Estas características se adecúan muy bien a los elementos que ofrece la API de alto nivel *Screen*, además del hecho de buscar la máxima portabilidad en el rango de todos los dispositivos que soportan MIDP; es por ello que se decidió que la vista se implementaría usando esta API.

3.4.2.6 Ofuscación y optimización de la aplicación

Al tratarse de una aplicación que se ejecuta en terminales móviles, se va a disponer de unos recursos limitados, por lo que cualquier medida que ayude a aligerar la aplicación merece ser tenida en cuenta. Una vez desarrollado por completo el cliente, se aplicó un proceso de ofuscado y optimización del código utilizando ProGuard, utilidad integrada dentro del entorno NetBeans.

Tras el proceso de ofuscación y optimización, se redujo la base de código en un 30% (sin tener en cuenta el peso de las imágenes que, como es lógico, no se ve afectado por este proceso), lo que redundó en una mayor velocidad y menor consumo de memoria por parte de la aplicación una vez desplegada en los terminales móviles.

3.4.3 SERVIDOR

3.4.3.1 Introducción

En la aplicación desarrollada, el grueso de la inteligencia del sistema se encuentra en el servidor. Éste ha de saber el procesamiento que tiene que realizar cuando recibe una petición de un cliente, y los datos que tiene que enviarle para que pueda seguir su funcionamiento correcto. Asimismo, también ha de llevar cuenta de las partidas que están en juego, ya sean de un jugador o varios, y en el caso de que haya alguna partida multijugador, deberá llevar también un control de acceso a los recursos compartidos, es decir, al modelo del sistema. Se puede ver un diagrama de clases del servidor en la Figura 3.6.

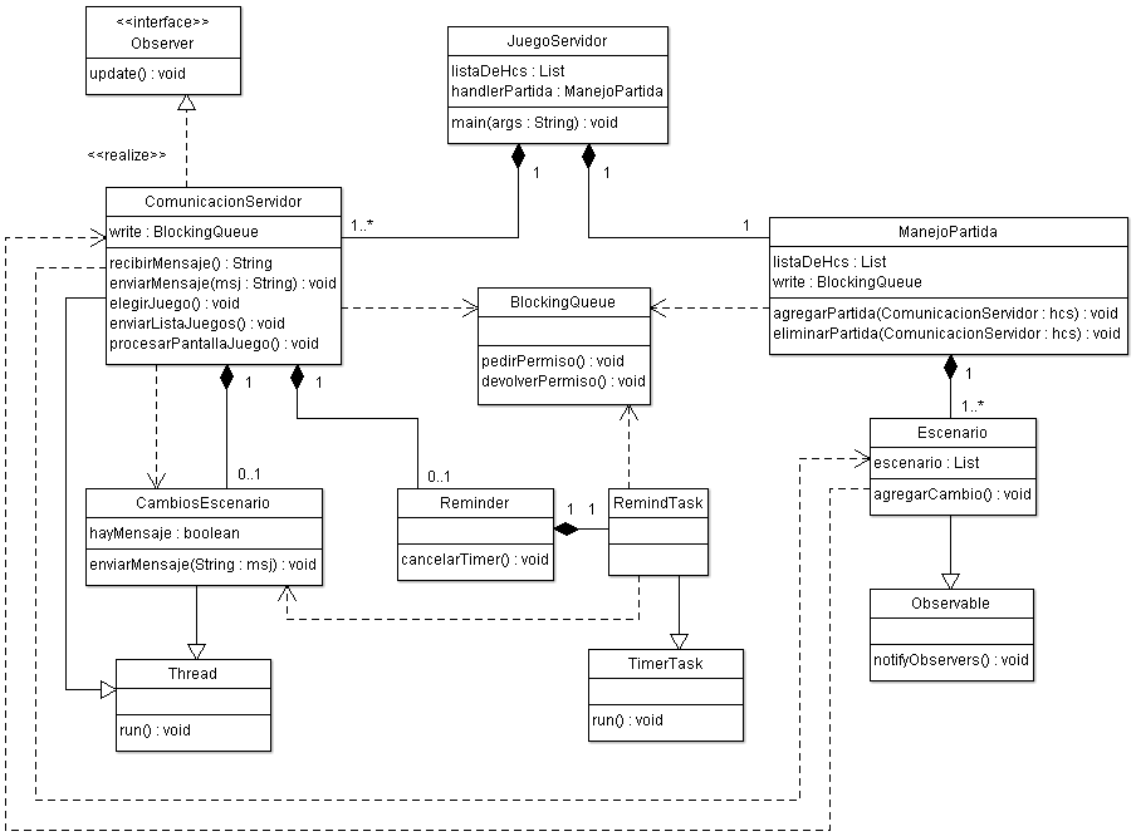


Figura 3.6 Diagrama de clases del servidor

En primer lugar está *JuegoServidor*, que es la clase principal en el lado del servidor, como indica la presencia del método *main()*. *JuegoServidor* se va a encargar de lanzar una instancia de *ComunicacionServidor* por cada usuario de la aplicación que llegue. *ComunicacionServidor* forma parte del módulo de comunicaciones del servidor, y a su vez tiene los métodos que proveen de inteligencia al sistema, y que realizan el procesado de las acciones que envían los clientes sobre el modelo del sistema, para a continuación enviarles el resultado y que siga el flujo de juego.

JuegoServidor ha de llevar un control de todas las instancias de *ComunicacionServidor* que ha lanzando, sin embargo, en lugar de hacerlo por sí mismo, lo delega en un objeto de tipo *ManejoPartida*, que básicamente va a ser el controlador encargado de mantener la estructura en la que se almacenan las partidas individuales y colectivas que hay, y dentro de las colectivas, los usuarios que hay jugando en cada una de ellas que comparten recursos, así como los testigos que permiten el acceso a estos recursos.

Este objeto de tipo *ManejoPartida* realiza el control por medio de una lista de acceso síncrono llamada *listaDeHcs*, en la que se guardan las referencias a todas las instancias de

ComunicacionServidor. Además, lleva el control relativo a la inserción y eliminación de partidas por medio de los métodos *agregarPartida()* y *eliminarPartida()*, así como a la asignación de permisos de escritura en archivos compartidos por varios usuarios.

ComunicacionServidor contiene el núcleo de la inteligencia del sistema. Es la clase que recibe una serie de opciones del cliente, realiza un procesado basándose en ellas y en el contenido del modelo del sistema, y genera un resultado que reenvía al cliente junto con las modificaciones del escenario en el que estaba. Esta clase contiene una serie de métodos gemelos a los de *ComunicacionCliente*, ya que están diseñados para interactuar entre ellos e intercambiarse información, aunque siempre con el procesado pesado cayendo de lado de los métodos correspondientes al servidor. Por ejemplo, se puede ver con los métodos *elegirJuegoCliente()*, que reside en *ComunicacionCliente*, y *elegirJuego()*, que reside en *ComunicacionServidor*.

El método en el que se concentra el bucle principal de juego y todo el procesado que éste requiere se denomina *doAction()*, y está situado en *ComunicacionServidor*, y a su vez dentro del método principal que contiene el bucle de juego, *procesarPantallaJuego()*. A este método le llega la acción que ha elegido tomar el usuario, así como los objetos o personajes con los que quiere realizar esta acción, y después de realizar un procesado basado en estos datos y el contenido de los archivos XML del modelo, actualiza dicho modelo, y extrae la nueva localización o la situación cambiada en la actual, que reenvía al cliente para que la presente al usuario y continúe el bucle de juego.

Una vez ejecutado *doAction()*, se ha realizado la acción elegida por el usuario, y se han calculado las consecuencias que ha tenido sobre el entorno. Antes de devolver el control al cliente, se realiza un recuento de los jugadores y NPC con los que se comparte escenario tras los cambios realizados, para así poder mostrarle al usuario, de forma actualizada, el conjunto de personajes (ya correspondan éstos a jugadores reales o a NPC controlados por la inteligencia del sistema) con los que puede interactuar. Esto se hace con los métodos *getUsersPantalla()* y *getNpcsPantalla()*.

3.4.3.2 Control de acceso

En el contexto de una partida multijugador, se tiene a un cierto número de usuarios de la plataforma compartiendo los archivos XML de configuración del juego y, en su caso,

modificándolos. Esto crea una serie de problemas que hay que resolver con un rígido esquema de control de acceso, que permita únicamente un acceso ordenado, de uno en uno, de los clientes al modelo.

Para ello se ha usado, como en el caso del cliente, una cola síncrona implementada en la clase *BlockingQueue*. El funcionamiento es similar, pero para mayor claridad semántica, en este caso se ha decidido renombrar los métodos de acceso para que se adecúen al trabajo que van a realizar en este contexto. Es por ello que se tendrán los métodos de acceso *pedirPermiso()*, que se encarga de intentar extraer un elemento de la cola, y si no lo hay, espera hasta que lo haya; y *devolverPermiso()*, que, por el contrario, intenta introducir un elemento en la cola. De este modo, se utilizará una cola síncrona llamada *write*, a modo de testigo, que tendrá que tener en su posesión la instancia de *ComunicacionServidor* que pretenda realizar una escritura en los archivos del modelo.

Debido a que las modificaciones a los archivos XML de configuración provienen en su gran mayoría de la parte que lleva la inteligencia del sistema, es decir, del método *procesarPantallaJuego()* y, más concretamente, de *doAction()* y los métodos a los que éste llama; se estimó oportuno proteger la ejecución de *doAction()* por medio de la invocación de *write.pedirPermiso()* justo antes de llamarlo, y cuando ya se ha ejecutado, realizar la devolución del testigo para que quede disponible para otros usuarios más adelante con *write.devolverPermiso()*. Se hizo lo mismo con el resto de métodos en el servidor que acceden a los recursos compartidos.

Por otra parte, la clase *Reminder* es la encargada de implementar un límite de tiempo en determinadas localizaciones. Para ello, contiene en su interior una clase *RemindTask* que hereda de *TimerTask*, en la que se define un tiempo de espera que ha de transcurrir antes de ejecutar su método *run()*. Este tiempo de espera será extraído del temporizador definido en la localización. Transcurrido ese tiempo, se ejecuta el método *run()* en el que se carga el siguiente temporizador definido si lo hay, a la vez que se le muestra al usuario el mensaje de alerta. Si no hay otro temporizador definido es porque el actual es el último, por tanto habrá que calcular las consecuencias que correspondan e informar al usuario.

El proceso para recuperar los temporizadores definidos es el siguiente: cuando se accede a una nueva localización, se comprueba si dicha localización tiene una limitación de tiempo definida. Si es así, se accede a los temporizadores definidos en los archivos

XML, para saber cuánto tiempo ha de pasar antes de notificar al usuario. Por tanto, también aquí habrá que proteger el acceso a los recursos compartidos, y es por ello que cuando expira el temporizador y se ejecuta el método *run()* de *ReminderTask*, se pide el testigo de acceso. Al final de este método ya se han calculado las consecuencias que ocurren si era el último temporizador definido, o se ha mostrado el aviso al usuario y se ha lanzado un nuevo temporizador. Es en este punto cuando se devuelve el testigo de acceso para que se pueda seguir con el flujo del programa. Es de notar que si el usuario abandona la localización en cualquier momento, es posible eliminar el temporizador por medio del método *cancelarTimer()* de *Reminder*.

Asimismo, también fue necesario añadir aquí la protección del testigo para que no ocurriesen colisiones a la hora de realizar las inserciones y eliminaciones de una partida en la lista general de partidas que lleva el controlador *ManejoPartida*. Es por ello que las llamadas a estos métodos de agregar y eliminar partida en el controlador van flanqueadas por una petición de testigo, y, a continuación cuando ya se han ejecutado, de su devolución para que el programa pueda seguir su curso normal de ejecución.

Cabe destacar que esta protección en el acceso a recursos compartidos sólo tiene sentido implementarla en el caso de que se esté jugando una partida multijugador, ya que será en ese caso en el único en el que pueda darse la eventualidad de que sea necesario regular el acceso de más de un cliente al mismo conjunto de archivos al mismo tiempo. Asimismo, también es importante reseñar que este testigo que protege el acceso a recursos compartidos no es global entre todos los clientes que se encuentren jugando en un momento determinado, sino que son todos los clientes que comparten una misma partida (y, por tanto, un mismo conjunto de archivos de configuración) los que comparten un mismo testigo.

3.4.3.3 Coherencia del modelo

El establecimiento de un orden en el acceso a los recursos compartidos, y la exigencia de que sólo haya un cliente modificando esos recursos en un determinado momento, son medidas completamente necesarias en el contexto de una aplicación que permite el acceso a esos recursos a varios clientes de forma asíncrona. Sin embargo, no es suficiente.

Cada uno de los clientes que ejecutan el motor de videojuegos, tiene en todo momento una copia local correspondiente a un fragmento de los archivos de configuración. Dada la

naturaleza por turnos de los videojuegos de aventura conversacional, un cliente podría modificar el modelo, y mientras tanto el resto de clientes que comparten localización con él tendrían una versión anticuada de esa localización, lo cual es peligroso. Por ejemplo, si la localización es una habitación, en la que hay una llave en el suelo, y hay dos jugadores que se encuentran en dicha habitación. Uno de ellos podría decidir coger la llave, para lo que seleccionaría la acción "coger", que se enviaría al servidor y éste realizaría el procesado necesario para establecer en los archivos de configuración que la llave pertenece a ese usuario, y a continuación le devolvería el control a dicho usuario, ya con la llave en su inventario. Pero, ¿qué pasaría con el otro usuario que estaba en la misma pantalla? Puesto que no ha hecho ninguna acción, no ha realizado ningún acceso al servidor, por lo que la copia local de los datos que tiene en su posesión le dice que la llave todavía está en el suelo. Esto podría llevar al usuario a intentar coger un objeto que ya no está en ese lugar, y podría dar lugar a consecuencias imprevistas (que le sustrajese la llave al otro jugador de su inventario, que se duplicase la llave y cada uno acabase con una, o incluso provocar un error crítico en la aplicación).

Es por ello que es imprescindible crear un mecanismo que permita garantizar la coherencia de los datos del modelo y eliminar discrepancias, esto es, que todos los clientes que compartan una partida (y por tanto comparten acceso a un modelo en común) tengan en todo momento la misma versión de esos datos.

Para ayudar a implementar esta garantía se ha utilizado el patrón observador. Este patrón permite que una serie de objetos observadores registren su interés en los cambios de otro objeto observable o sujeto. En el momento en que haya alguna modificación en el sujeto, los observadores son notificados para que actúen en consecuencia.

En el servidor desarrollado, se tiene un objeto *Escenario*, que hereda de la clase abstracta *Observable* de Java. Este objeto *Escenario* sería el sujeto, cuyos cambios van a interesar a todos los usuarios que comparten una partida. Por su parte, todas las instancias de *ComunicacionServidor* implementan el interfaz *Observador* de Java. Esto quiere decir que tienen que implementar obligatoriamente un método *update()*, que es al que se llama cuando hay cambios en el sujeto, en este caso *Escenario*.

Por tanto, conforme se van añadiendo nuevos usuarios a una partida ya existente, se forma un conjunto de instancias de *ComunicacionServidor*, cada una de las cuales

corresponde a un usuario, que comparten el testigo de acceso a recursos compartidos *write* y un conjunto de archivos de configuración. Al irse añadiendo estos usuarios a la partida, *ManejoPartida* va registrando su interés en el sujeto *Escenario*. Cuando uno de los usuarios quiere realizar una acción, y por tanto, provocar modificaciones en el modelo, en primer lugar se protege su acceso único y ordenado por medio del uso del testigo *write*. Y a continuación, se le permite realizar los cambios oportunos en el modelo, guardar dichos cambios, y se llama al único método de *Escenario*, *agregarCambio()*. Este método recibe varios datos, entre ellos el usuario que ha realizado el cambio, la pantalla en la que se encuentra ese usuario, el tipo de cambio (si ha cambiado algún objeto, si ha enviado un mensaje de chat a otro usuario, etc.), y otros datos.

Desde *agregarCambio()* se llama a *notifyObservers()*, que es el método que provee la clase *Observable* y que permite notificar a todos los usuarios que hayan declarado su interés en el sujeto, que ha habido cambios en el mismo. Por tanto, inmediatamente tras producirse los cambios en el modelo, todos los usuarios son notificados de que ha habido cambios, y en su método *update()* reciben encapsulados los datos que recibió *agregarCambio()*, y que son necesarios para saber si se ha visto comprometida la coherencia de los datos de que disponen. Así, se considera que la coherencia de los datos del modelo se ha visto comprometida y es necesario volver a pedir al servidor su reenvío si un usuario ha realizado cambios en la misma pantalla en la que nos encontramos, y si ese usuario no somos nosotros (ya que el método *update()* del propio usuario que ha provocado los cambios también es llamado, porque es un observador al igual que todos los demás). Es necesario realizar estas comprobaciones porque no es conveniente estar interrumpiendo constantemente al resto de usuarios por los cambios que pueda estar haciendo uno de ellos, si no va a tener repercusión en la coherencia de sus datos.

Si tras esas comprobaciones se ve que, en efecto, hay que realizar una actualización del estado de la pantalla en el cliente, desde el propio método *update()* se notifica a *CambiosEscenario*, que se encarga de mantener una conexión con el cliente por medio de un socket distinto al principal, para que no haya intromisiones en el flujo de juego. Por medio de *CambiosEscenario*, se envía al cliente la notificación de que tiene que iniciar una actualización de sus datos. Esto se recibe en el cliente en una instancia de *RecibeCambiosEscenario* que, a su vez, notifica a la instancia de *ComunicacionCliente*, que pide al servidor un reenvío de los datos, asegurando de esa forma que se tiene la última versión de éstos en todo momento.

Es de notar que este mecanismo también se utiliza para notificar a los usuarios de otros eventos y no sólo de la posibilidad de que la coherencia de los datos no esté garantizada, por ejemplo para enviar un mensaje de chat de un usuario a otro (con la opción "hablar con" ese usuario), se notifica del mismo modo, lo que permite que el mensaje llegue al destino instantes después de ser enviado en lugar de tener que esperar al siguiente turno en el que el receptor del mensaje realice alguna acción y, por tanto, acceda a los datos actualizados en el servidor.

Sin embargo, a pesar de la protección que proporciona este mecanismo de actualización de los datos locales en el cliente, también será necesario prever los posibles problemas que puedan surgir cuando ocurren dos cambios en el modelo de forma casi simultánea, de modo que no de tiempo a actualizar el modelo para el resto de jugadores de la pantalla, por ejemplo si dos usuarios intentan coger un mismo objeto a la vez. En este caso, el uso del testigo *write* hará que, necesariamente, un usuario acceda antes que el otro al objeto, y será éste quien lo recoja. En cuanto al otro usuario, para evitar problemas, siempre se comprobará en el servidor si se está en una partida multijugador que el objeto o usuario sobre el que se va a aplicar una acción seleccionada en el móvil, sigue ahí antes de efectuar dicha acción. Al realizar esa consulta justo antes de llevar a cabo la acción se puede detectar si el objeto o usuario sobre el que se iba a realizar la acción ha desaparecido del escenario y todavía no ha dado tiempo a que se provoque la actualización de los datos en el cliente. En tal caso, se mostrará por pantalla un mensaje de aviso indicando que el objeto o usuario ya no están en esa localización, y es por ello imposible realizar la acción que se había elegido.

Asimismo, también se usa este mecanismo para las peticiones de actualización asíncronas del servidor cuando se está en una partida individual. En este caso, no tiene sentido proteger el acceso a unos recursos compartidos porque se tiene acceso exclusivo, ni tampoco hay que asegurar la coherencia del modelo por la misma razón. Sin embargo, hay que implementar una forma de que el servidor pueda avisar al cliente de que tiene que realizar una actualización del contexto de juego, por ejemplo en el caso de que haya expirado un temporizador definido para esa localización. En ese caso, basta con definir un nuevo tipo de actualización para que sólo afecte a un jugador, ya que los temporizadores se definen de forma local al usuario, y seguir el proceso descrito.

3.4.3.4 Estructura multihilo

Este proyecto hace uso de los hilos de ejecución o *threads*. Los programas multihilo hacen un uso de la multitarea que les permite hacer parecer que están ejecutando varias tareas de forma simultánea. Todas las clases que vayan a ejecutar tareas de red, por ejemplo abriendo y manteniendo una comunicación por medio de sockets, es necesario que estén planteadas como hilos de ejecución, ya que el acceso a la red provoca unos retardos varios órdenes de magnitud superiores a los que puede haber dentro de una máquina física. Es por ello que es necesario que estos accesos sean realizados desde hilos, que pueden quedarse a la espera, y cuando reciban información desde la red, volver a despertarse y seguir ejecutando el flujo de la aplicación.

Además, las características de los hilos permiten que sea sencillo realizar una creación dinámica de estos conforme vaya haciendo falta. En el caso del servidor implementado, cada vez que se recibe una petición de un nuevo cliente que desea conectarse con el servidor, se lanza un nuevo hilo de ejecución *ComunicacionServidor*. Si más adelante ese cliente decide entrar en una partida multijugador, y así compartir el terreno de juego con varios usuarios de forma simultánea, se lanzará un hilo de ejecución de *CambiosEscenario* para permitirle recibir las notificaciones de cambios realizados en el modelo.

Si el servidor pierde la conexión con los clientes, se producirá en cada uno de los hilos del servidor *ComunicacionServidor* que atienden a cada cliente una excepción de tipo *IOException*. Al ocurrir esto, se realiza una limpieza de los hilos que hayan causado esa excepción (que serán todos si, por ejemplo, se ha perdido la conexión a Internet del servidor) y desde *JuegoServidor* se seguirá en el bucle de conexión de nuevos clientes esperando a que éstos lleguen en cuanto se restablezca la conexión. Así, no será necesario volver a arrancar el servidor en el evento de una pérdida de la conexión.

Cuando el cliente abandona el juego, los hilos creados para darle servicio son eliminados por el recolector de basura automático de Java, para así reducir el consumo de memoria global de la aplicación.

3.4.3.5 Optimización del tráfico

Debido a tratarse de una aplicación que va a ser distribuida en teléfonos móviles que van a usar una conexión inalámbrica para acceder a los videojuegos de la plataforma, el

tamaño de las transferencias de datos realizadas entre servidor y clientes va a ser importante, principalmente por dos motivos: el tiempo de espera de los usuarios y el coste. Un usuario tiene que soportar un determinado tiempo de espera desde que realiza una acción hasta que se encuentra con que han terminado los intercambios con el servidor y puede continuar jugando. Este tiempo de espera deberá ser lo más pequeño posible para no causar frustraciones al usuario, al no poder ejercer durante esos instantes control directo sobre lo que ocurre en el videojuego. En cuanto al coste, en el caso de las alternativas de conexión a la plataforma de videojuegos que lleven un coste de conexión asociado, es obvio que será deseable que el tamaño de las transferencias de datos con el servidor sea lo menor posible.

Para intentar optimizar el tamaño de las transferencias realizadas entre servidor y clientes se tendrá en cuenta que los elementos que más ocupan son las imágenes que van a ilustrar las localizaciones. Por ello, se mantendrá en los clientes en un *buffer* la última imagen transferida del servidor, y en el caso de que la acción que quiere realizar el usuario no implique un cambio de localización, se aprovechará dicha imagen ahorrando así la transferencia de una nueva y, con ello, la mayor parte del tráfico que se produce al cargar una localización de juego.

3.4.4 PROTOCOLO IMPLEMENTADO

3.4.4.1 Introducción

Con el fin de llevar a cabo el presente proyecto, y teniendo clara la estructura que habían de tener tanto el cliente como el servidor, quedaba pendiente la no menos importante tarea de definir un protocolo de comunicaciones que hiciese que cliente y servidor se entendiesen entre ellos, y que pudiera formarse un flujo de programa que englobara a ambos extremos por medio de la comunicación establecida por el protocolo.

Para ello, se consideró un protocolo sencillo, que realizase un envío de mensajes en forma de cadenas de texto. Estos mensajes son analizados por el otro extremo de la comunicación, y así pueden saber el estado en el flujo de programa en el que el emisor se encuentra, y por lo tanto saber cómo ha de comportarse para establecer una comunicación con éste. Tras el mensaje que hace de cabecera e indica al otro extremo de la comunicación el punto de la máquina de estados en el que se encuentra, se enviarán los datos correspondientes a ese estado.

Hay dos variedades de mensajes: los que se van a enviar siguiendo el flujo normal de juego entre *ComunicacionCliente* y *ComunicacionServidor*, y los que se van a enviar a modo de petición de actualización asíncrona desde *CambiosEscenario* en el servidor a *RecibeCambiosEscenario* en el cliente.

3.4.4.2 Mensajes de flujo de juego

La primera variedad de mensajes va a tener un orden establecido que emana del orden inherente a los procesos del juego, por ejemplo tendrá que enviarse el mensaje de que el usuario se va a identificar antes que el mensaje de que ha elegido realizar una acción, ya que para realizar una acción el usuario tiene que estar identificado y jugando. Por ello, estos mensajes llevarán un identificador numérico que indique el orden en que se envían. Asimismo, para mayor claridad, se incluirá también en el propio mensaje si es un mensaje enviado por el cliente o por el servidor. Se tendrán los siguientes mensajes de este primer tipo:

```
“MENSAJE S 1 ENVIO_LISTA_JUEGOS”
“MENSAJE C 2 ENVIO_JUEGO_ELEGIDO”
“MENSAJE S 3 HAZ_LOGIN”
“MENSAJE C 4 ENVIO_DATOS_LOGIN”
“MENSAJE S 5 ENVIO_TEXTOS_RESTART”
“MENSAJE C 6 ENVIO_ELECCION_RESTART”
“MENSAJE S 7 ENVIO_PANTALLA_JUEGO”
“MENSAJE C 8 ENVIO_ELECCIONES”
“Fin”
```

Como se puede ver, todos los mensajes (excepto el último) tienen una estructura similar. Comienzan por la palabra “MENSAJE” seguido de la letra “C” si dicho mensaje corresponde al vocabulario de mensajes del cliente, o de la letra “S” si dicho mensaje pertenece al vocabulario de mensajes del servidor. Es decir, los tipos de mensaje antes comentados son fijos, no se crean sobre la marcha, y sólo existen los nueve tipos explicados. Por ejemplo, no podrá existir un hipotético “MENSAJE S 8 ENVIO_ELECCIONES”, ya que el único que envía las elecciones es el cliente.

Tras esto, se incluye el número de orden del mensaje. Hay que decir que el intercambio de mensajes no se tiene que realizar necesariamente en el orden en el que indican sus identificadores numéricos (por ejemplo una vez se ha enviado el mensaje 7 desde el servidor y el cliente contesta con el mensaje 8, se podrá enviar otro mensaje de tipo 7 desde el servidor, al que el cliente puede contestar con un nuevo mensaje de tipo 8), ni

tampoco hay que enviarlos todos (por ejemplo si el usuario no quiere reiniciar su partida, no se envían los mensajes 5 y 6). Pero sí que ayudan a indicar un orden aproximado de cómo se llevan a cabo los procesos.

Hay un tipo especial de mensaje dentro de este vocabulario, que es el mensaje “Fin”. Este mensaje es distinto a los demás para hacerlo destacar, y será enviado únicamente desde el cliente al servidor cuando el jugador decida que quiere abandonar la partida. El mensaje “Fin” no irá acompañado de información adicional.

El protocolo tanto en el cliente como en el servidor se basa en un bucle que recibe un mensaje del otro extremo acompañado de unos ciertos datos, realiza sus actividades (interaccionar con el usuario en el caso del cliente, hacer un procesado de los datos recibidos en el caso del servidor) y envía un mensaje con la siguiente acción a realizar, así como los datos pertinentes. Para poner en marcha esta máquina de estados, el servidor envía un primer mensaje al cliente acompañado de unos datos, y a partir de ahí, cada uno de los dos extremos recibe un mensaje acompañado de unos datos, realiza sus actividades con estos datos, y envía un mensaje para indicar al otro extremo cómo tratar lo que, a continuación, le va a enviar.

El flujo detallado del protocolo se puede ver en la Figura 3.7. En ella se indica el mensaje que se envía en el protocolo, seguido del intercambio de información que se realiza entre cliente y servidor. Bajo cada grupo de transacciones se señala el sentido en el que se llevan a cabo, C → S indica sentido del cliente al servidor, y C ← S indica sentido del servidor al cliente. A continuación se detalla el flujo del protocolo:

1. El cliente se conecta al servidor, e inmediatamente se pone en espera de recibir un mensaje por parte del servidor. En el momento que la conexión está establecida, el servidor le envía al cliente un primer mensaje de tipo “MENSAJE S 1 ENVIO_LISTA_JUEGOS” acompañado de la lista de videojuegos disponibles en la plataforma en ese instante. Para enviar la lista de videojuegos, primero se envía el número de videojuegos de la plataforma, y a continuación una cadena de texto por cada uno de los videojuegos que haya. Esto pone en marcha la máquina de estados.

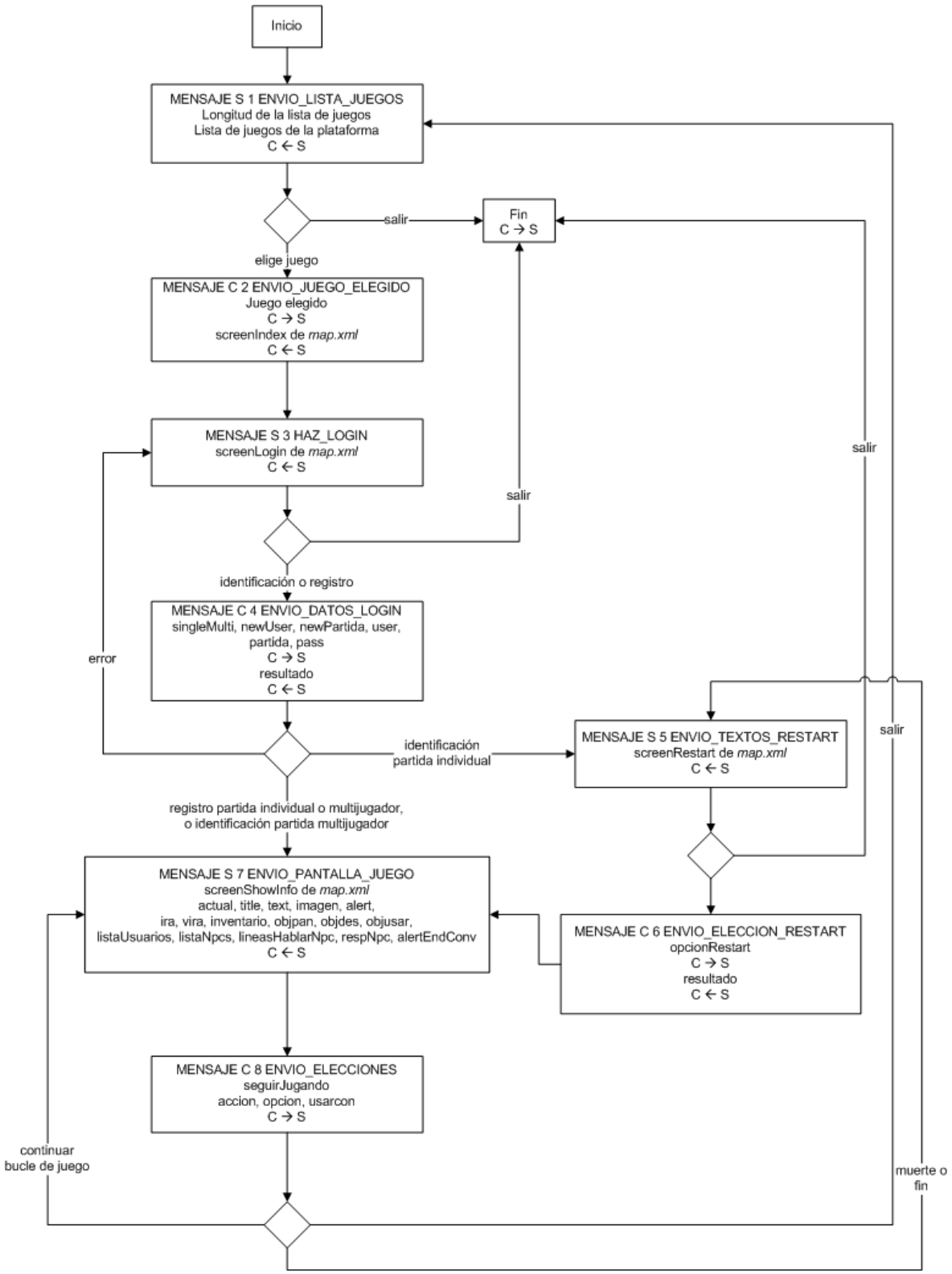


Figura 3.7 Diagrama de estados del protocolo

2. Al cliente le llega este mensaje, lo procesa y recibe la lista de juegos para mostrarla al usuario. Hecho esto el usuario puede seleccionar un videojuego de la lista, caso en el que el cliente enviará un mensaje de tipo "MENSAJE C 2

ENVIO_JUEGO_ELEGIDO" acompañado del nombre del videojuego elegido. O bien puede elegir salir de la aplicación, caso en el que se enviará el mensaje "Fin" al servidor para indicar que se quiere abandonar la aplicación.

3. El servidor recibe y analiza el mensaje. Si es un mensaje "Fin" realizará las labores necesarias para eliminar la partida. Si, en cambio, es un mensaje de tipo "MENSAJE C 2 ENVIO_JUEGO_ELEGIDO", el servidor recibirá a continuación el nombre del videojuego seleccionado por el cliente. Tras esto, el servidor enviará al cliente el fragmento "<screenIndex>" del archivo *map.xml* para que pueda construir los formularios de presentación del videojuego elegido. Acto seguido, enviará un mensaje de tipo "MENSAJE S 3 HAZ_LOGIN" y, a continuación, el fragmento "<screenLogin>" de *map.xml*, en el que está la información necesaria para construir los formularios de identificación y registro de usuarios/partidas.

4. El cliente recibe este último mensaje y se prepara para recibir los datos para crear los formularios de identificación y registro. Una vez recibidos, el usuario puede elegir salir de la aplicación, con lo que se enviaría el mensaje "Fin" al servidor que procedería como en el punto 3, o bien puede decidir introducir su información de identificación o registrar una nueva partida individual o multijugador. Si se elige esto último el usuario navega por los formularios de registro o identificación hasta haber introducido todos sus datos, momento en el cual enviará un mensaje de tipo "MENSAJE C 4 ENVIO_DATOS_LOGIN" acompañado de sus datos de identificación o registro, formados por las siguientes cadenas de texto:
   - *singleMulti*, que indica si el usuario ha elegido una partida individual o colectiva. Puede tomar los valores "single player" o "multi player".
   - *newUser* y *newPartida*, que indican si el usuario ha elegido identificarse en una partida individual (*newUser* sería "no") o colectiva (*newPartida* sería "no") ya existente, o bien crear una nueva partida (si es individual, *newUser* sería "yes", y si es colectiva, *newPartida* sería "yes").
   - *user*, que indica el nombre de usuario individual elegido por el jugador.
   - *partida*, que indica el nombre de partida multijugador elegida por el usuario.
   - *pass*, que indica la contraseña elegida por el usuario.

   Tras esto, recibirá una respuesta del servidor indicándole cómo transcurrió el proceso de identificación o registro, para lo que se envía la variable *resultado*. Esta variable puede tomar los valores:

- “Nuevo usuario OK”, si se crea una nueva partida individual o multijugador de forma correcta.
- “Error al crear el directorio”, si se introduce algún carácter extraño como nombre de usuario o partida y el sistema operativo del servidor no puede crear un directorio con ese nombre.
- “El usuario ya existe”, si se intenta crear un usuario individual con un nombre ya registrado.
- “Usuario validado OK”, si se ha identificado en una partida individual correctamente.
- “Usuario no encontrado”, si se intenta identificar con un nombre de usuario que no está registrado, o con una contraseña que no corresponde al nombre de usuario introducido.
- “Esa partida ya existe”, si se intenta crear una partida multijugador con un nombre ya registrado.
- “Partida validada OK”, si se ha identificado en una partida multijugador correctamente.
- “Partida no encontrada”, si se intenta entrar a una partida que no está registrada, o con una contraseña que no corresponde al nombre de partida introducido.

5. El servidor recibe este mensaje acompañado de los datos de identificación o registro del usuario, y procede a realizar la identificación o registro con ellos. Una vez hecho esto, es posible que se haya realizado correctamente, o bien que haya habido algún problema (por ejemplo que se haya intentado registrar un nombre de partida ya existente, o que se haya intentado realizar la identificación con una contraseña que no corresponde con el nombre de usuario introducido). Se envía al usuario un mensaje indicándole cómo transcurrió el proceso. En base a como haya transcurrido se pueden tener varias situaciones en el servidor:
   a) Si se trata de un usuario individual que se identifica correctamente, se enviará un mensaje al cliente del tipo “MENSAJE S 5 ENVIO_TEXTOS_RESTART” seguido del fragmento “<screenRestart>” de *map.xml* que se usará para construir un formulario con el que se le dará la opción de continuar la partida por donde la dejó anteriormente o bien reiniciarla para comenzar a jugar desde el principio. El proceso sigue en el paso 6.

b) Si se trata de un usuario individual que realiza el registro correctamente o de un usuario de partida multijugador que se identifica o realiza el registro correctamente, se tiene todo lo necesario para comenzar el juego, por lo que se enviará al cliente un mensaje de tipo "MENSAJE S 7 ENVIO_PANTALLA_JUEGO" seguido del fragmento "<screenShowInfo>" de *map.xml*, que contiene los textos a mostrar en los formularios que se usan en el bucle principal de juego. Tras esto, se enviarán una serie de cadenas de texto que contienen todos los datos necesarios para construir la localización a mostrar al usuario:

- *actual*, donde se indica el identificador de la localización actual.
- *title*, que contiene el título de la localización actual, a mostrar en el encabezado del formulario de juego.
- *text*, que contiene la descripción de la localización que se muestra al usuario.
- Para el envío de la imagen, en primer lugar se envía una cadena de texto en la que se dice si la imagen que hay que mostrar es la misma que ya hay cargada y por tanto se puede aprovechar ("misma imagen a cargar") o no lo es y hay que enviar la nueva imagen desde el servidor ("nueva imagen a cargar"). En el caso de que sea necesario enviar una nueva imagen del servidor, primero se envía la longitud que va a tener la imagen para conocer el tamaño del *buffer* que hay que reservar, y a continuación se envía la propia imagen como un flujo binario de esa longitud.
- *alert*, que contiene un mensaje de información que puede enviar el servidor al cliente cuando realiza una acción.
- *ira*, que consiste en la lista de direcciones a las que puede moverse el jugador con la opción "Ir a", concatenadas en una única cadena de texto.
- *vira*, que está formada por los valores numéricos de las direcciones contenidas en *ira*, concatenados como en el caso anterior.
- *inventario*, que contiene la lista concatenada de objetos que forma el inventario.
- *objpan*, que contiene la lista concatenada de objetos de la pantalla actual.
- *objdes*, que contiene la descripción que se muestra de los objetos que hay en la pantalla actual.
- *objusar*, que contiene la lista concatenada de objetos a mostrar en la opción "Usar con".

- *listaUsuarios*, que contiene la lista concatenada de jugadores en una partida multijugador que se encuentran en la localización actual.
- *listaNpcs*, que contiene la lista concatenada de NPC que se encuentran en la localización actual.
- *lineasHablarNpc*, que contiene la lista concatenada de las frases que puede decir un usuario al NPC para el nivel de la conversación en el que se encuentren, en el caso de que haya un NPC en esa pantalla.
- *respNpc*, que contiene la respuesta del NPC a la última frase dicha por el usuario, en el caso de que la haya recibido.
- *alertEndConv*, que contiene el mensaje a mostrar al usuario cuando ha llegado al fin de la conversación con un NPC, en el caso de que haya ocurrido.

Las listas concatenadas consisten en una serie de elementos delimitados por un separador, para así poder realizar el envío en una única cadena de texto, y separar en el cliente cuando se recibe.

Si el personaje ha muerto o ha finalizado el juego, sólo se enviarán *actual*, *title*, *text*, la imagen correspondiente y *alert*, al no ser necesarias las demás variables. El proceso sigue en el paso 7.

c) En cualquier otro caso, es decir, si ha habido problemas, se enviará un mensaje del tipo “MENSAJE S 3 HAZ_LOGIN” así como el fragmento “<screenLogin>” de *map.xml*, para que el usuario intente de nuevo la identificación o el registro, por lo que se procedería en el cliente como se ha descrito en el punto 4.

6. A este estado se llega si el servidor envió un mensaje de tipo “MENSAJE S 5 ENVIO_TEXTOS_RESTART”. El cliente recibe este mensaje así como el fragmento “<screenRestart>” de *map.xml* que contiene la información para mostrar al usuario las opciones de continuar por donde estuvo jugando o reiniciar la partida. En este punto el usuario puede decidir abandonar la partida, caso en el que se enviará el mensaje “Fin” al servidor, que lo tratará como en el paso 3. Por el contrario, si elige una de las dos opciones que se le presentan, se enviará al servidor un mensaje del tipo “MENSAJE C 6 ENVIO_ELECCION_RESTART” acompañado de la variable *opcionRestart* que indica si quiere poner a cero sus progresos (caso en el que tomará el valor “restart”) o no (caso en el que tomará el valor “continuar”). Si quiere hacerlo, el servidor modificará el modelo para

devolverlo a su estado inicial. Se enviará la variable *resultado* al cliente, pero en este caso al tratarse de una partida individual ya validada anteriormente, siempre va a ser "Usuario validado OK". Hecho esto, ya se tendrá todo lo necesario para comenzar el juego, por lo que el servidor enviará un mensaje del tipo "MENSAJE S 7 ENVIO_PANTALLA_JUEGO" seguido de los datos de la localización, como se comentó en el paso 5 b).

7. En este estado el cliente ha recibido el mensaje de tipo "MENSAJE S 7 ENVIO_PANTALLA_JUEGO" seguido de los datos de la localización que tiene que mostrar al usuario. Una vez mostrada la situación al usuario, se espera a que éste elija hacer una acción o abandonar la aplicación. Hecha la elección, el cliente envía un mensaje de tipo "MENSAJE C 8 ENVIO_ELECCIONES" seguido de:
   - *seguirJugando*, que puede tomar los valores "salir" (cuando el usuario ha decidido abandonar el juego), "recomenzar" (cuando el jugador ha muerto o finalizado el juego con éxito, y decide comenzar de nuevo el juego) o "ok" (cuando el usuario desea seguir jugando y su personaje no ha muerto ni finalizado el juego).

   Si *seguirJugando* toma el valor "ok", a continuación se enviarán las variables:
   - *accion*, en la que se indica la acción que se ha elegido de la lista de acciones.
   - *opcion*, en la que se indica el lugar al que se va a mover el usuario (si la acción elegida es "Ir a") o el objeto elegido sobre el que se va a aplicar la acción (si se eligió "Coger", "Dejar", "Mirar", "Usar", "Abrir", "Cerrar", "Usar con", "Hablar con" o "Dar a").
   - *usarcon*, en la que se indica el segundo objeto elegido con la opción "Usar con", el texto enviado por el usuario si se ha elegido "Hablar con" un usuario de partida multijugador, o el usuario o NPC que va a recibir el objeto si se ha elegido "Dar a".

8. El servidor recibe el mensaje y la información adicional sobre si el usuario quiere seguir jugando y si es necesario recomenzar la partida. Dependiendo de esta información, se podrán tener varios casos:
   a) Si el usuario ha elegido salir se eliminarán los datos de manejo de la partida en el servidor y se le enviará a la lista de videojuegos, por lo que el servidor envía un mensaje de tipo "MENSAJE S 1 ENVIO_LISTA_JUEGOS" seguido de la lista de videojuegos de la plataforma. El proceso continuaría en el paso 2.

b) Si es necesario recomenzar la partida porque el jugador haya muerto o porque se ha llegado al final del videojuego, hay dos opciones. Si la partida es individual se envía desde el servidor un mensaje de tipo "MENSAJE S 5 ENVIO_TEXTOS_RESTART" seguido de la información para construir el formulario de reinicio, y se continuaría por el paso 6. En cambio, si la partida es multijugador, se sitúa al jugador en la pantalla inicial (porque no tendría sentido reiniciar una partida en la que hay más usuarios jugando en ese momento) y se envía desde el servidor un mensaje de tipo "MENSAJE S 7 ENVIO_PANTALLA_JUEGO" acompañado de la información de la localización que hay que mostrar al usuario, en este caso la inicial.

c) Si, por último, no hay que recomenzar la partida ni el usuario ha elegido salir, se continúa el flujo normal de juego. Para ello, se reciben en el servidor la acción que el usuario ha elegido así como los datos que la acompañan, se envía un mensaje del tipo "MENSAJE S 7 ENVIO_PANTALLA_JUEGO" al cliente para que se prepare para recibir la nueva localización que resulta de las acciones realizadas, y se realiza el procesado de dicha acción para calcular el nuevo contexto de juego, que se envía al usuario. Hecho esto, el proceso continúa por el paso 7, cerrando el bucle de juego.

3.4.4.3 Mensajes de petición de actualización

En el caso de la segunda variedad de mensajes, los que se van a enviar a modo de petición de actualización asíncrona desde el servidor al cliente, no existe un orden establecido, ya que se enviarán dependiendo de las acciones que realicen los usuarios y de otros parámetros como el número de jugadores presentes en la pantalla. Estos mensajes se envían siempre desde el servidor al cliente, por lo que tampoco es necesario incluir en el propio mensaje quién es el extremo de la comunicación que lo envía.

El funcionamiento general de este mecanismo de actualización se basa en enviar peticiones desde el servidor a los clientes en las que se les indica que tienen que realizar una recarga del contexto de juego. La estructura de petición de una nueva localización al servidor ya está especificada, como se vio en el apartado 3.4.4.2, por medio del envío de mensajes del tipo "MENSAJE C 8 ENVIO_ELECCIONES" a los que el servidor contesta con mensajes del tipo "MENSAJE S 7 ENVIO_PANTALLA_JUEGO". Por ello, solamente será necesario hacer que los datos que indican qué acciones ha realizado el

usuario sean nulos, de modo que el servidor entiende que realmente no se quiere realizar ninguna acción, sino la recarga de la localización actual.

Todos los mensajes asíncronos de actualización que recibe el cliente van a venir precedidos de un mensaje de tipo “MENSAJE ACTUALIZACION”, que siempre será el mismo e indica que se va a recibir un mensaje a continuación que indicará el tipo de actualización.

El vocabulario de mensajes que se pueden recibir tras el mensaje que actúa a modo de preámbulo es el siguiente:

“entrar a pantalla”
“salir de pantalla”
“hablar enviado”
“actualizar pantalla”
“juego terminado”
“timer”

Como en el caso anterior, se trata de cadenas de texto que se envían directamente por la red, sin modificaciones, tras lo cual se envían datos adicionales que el cliente necesita. Estas peticiones sólo se mandarán si los cambios ocurridos son relevantes al cliente, por ejemplo, si han ocurrido en la localización en la que se encuentra.

En el caso de que el servidor envíe un mensaje de tipo “entrar a pantalla” o “salir de pantalla”, a continuación se envía una cadena de texto con el nombre del usuario que ha causado dicha modificación. De este modo, el cliente realiza una recarga del contexto de juego en el que ya se encuentra el usuario que ha llegado a la localización, o ya no está el usuario que la ha abandonado. Además, se muestra por pantalla que el usuario que ha causado dicha actualización ha llegado o se va de dicha localización, respectivamente.

Si el mensaje recibido en el cliente es de tipo “hablar enviado” es porque otro jugador ha decidido enviarle un mensaje de texto. En ese caso, tras este mensaje se recibe el mensaje en cuestión como una cadena de texto, y se provoca una recarga del contexto de juego en el cliente, tras la cual ya se muestra por pantalla el mensaje que tenía al usuario como destinatario.

Si el servidor envía un mensaje de tipo “actualizar pantalla”, esto quiere decir que han ocurrido cambios en el modelo ocasionados por otro jugador, por lo que es necesario que todos los usuarios que estén en la localización en la que ha habido cambios realicen una recarga del contexto de juego. Tras esto, se recibe otro mensaje formado por la cadena de texto “actualizar”. Una vez hecho esto, el juego puede continuar.

En el caso de que el cliente reciba un mensaje de tipo “juego terminado” quiere decir que otro jugador en su partida colectiva ha logrado acabar el juego con éxito, por lo que se provoca una recarga del contexto de juego para que todos los usuarios puedan saber que ha ocurrido. Tras el mensaje de juego terminado se envía el nombre del usuario que ha acabado el juego como una cadena de texto, y se muestra por pantalla a todos los jugadores con los que comparte partida.

Por último, en el caso de que se reciba un mensaje de tipo “timer”, quiere decir que la localización en la que se encuentra el usuario tiene definido un temporizador que ha expirado, por lo que se le informa de que tiene que realizar una recarga para ver cuál es la nueva situación en la que se encuentra. Tras este mensaje se envía una cadena de texto que contiene un texto que se muestra al usuario, a modo de alerta, para avisarle de la eventualidad de que el temporizador ha expirado. Es de notar que este tipo de petición de actualización asíncrona es la única que ocurre en el caso de partidas individuales, ya que no existen otros usuarios que puedan provocar cambios en el modelo ni en el entorno de juego (por ejemplo, entrando o saliendo de la localización en la que se encuentra el usuario).

3.4.4.4 Sockets

La comunicación que establece el protocolo está basada en sockets. Los sockets son un mecanismo de programación de bajo nivel, que tienen la particularidad de que establecen un canal de comunicación con el otro extremo, y sólo envían la información que se les indica explícitamente (en este caso, los mensajes del protocolo), por lo que realizan un uso optimizado del ancho de banda disponible. Sin embargo, será necesario emplear una estructura de tipo *switch* (compuesta por varias condiciones *if – else if*) en recepción para poder hacer el tratamiento de mensajes, aunque esto no es un problema si, como es el caso, se tienen relativamente pocos tipos de mensajes distintos.

El protocolo contempla el uso de dos sockets por cada cliente. Inicialmente se establecerá una conexión bidireccional utilizando el primer puerto introducido. Por medio de esta conexión principal de juego será por la que se enviarán los mensajes de flujo de juego detallados en el apartado 3.4.4.2, así como la información que los acompaña. Más adelante, una vez que se ha recibido la primera localización de juego por parte del cliente, se abrirá otra conexión unidireccional del cliente al servidor, a través de la que se enviarán los mensajes de petición de actualización detallados en el apartado 3.4.4.3, así como los datos que los acompañan. En el caso de tener una partida individual sólo se producirá un tipo de petición de actualización, la correspondiente a un temporizador que ha expirado. En cambio, en una partida multijugador se podrán producir todas las variedades de peticiones de actualización, tanto las correspondientes a temporizadores que expiran como las que corresponden a acciones realizadas por otros usuarios que afectan a la localización en la que el usuario se encuentra.

Así, cuando se lance el servidor y se le diga en qué puerto ha de escuchar, realmente estará utilizando ese puerto y el siguiente, para poder ofrecer la funcionalidad de partidas multijugador y el establecimiento de un tiempo límite en determinadas localizaciones. En el cliente este proceso es totalmente transparente, de modo que simplemente tendrá que introducir la dirección IP y el puerto correspondiente al socket principal, y automáticamente ya se establecerá su conexión asociada. Cuando se haya recibido la primera localización de juego se establecerá otra conexión con el socket de actualizaciones, situado en el puerto consecutivo al primero introducido.

El acceso a los sockets tanto en el cliente como en el servidor se realizó haciendo uso de las primitivas de lectura y escritura de cadenas de texto sobre sockets que proporciona Java, para facilitar tanto el envío como la recepción de mensajes y variables. En el caso de los fragmentos de archivo XML y las imágenes, el envío se realizó directamente convirtiendo los recursos en un flujo de bytes, calculando su tamaño y enviando al otro extremo en primer lugar el número de bytes que iba a recibir, y acto seguido la información binaria.

3.4.4.5 Diagramas de funcionamiento del protocolo

En este epígrafe se van a analizar, por medio del uso de diagramas de secuencia, algunos casos representativos del flujo de programa en el que se vea el uso del protocolo de comunicación implementado.

Los diagramas de secuencia son una forma de comunicar la evolución de un sistema en el tiempo por medio de los mensajes intercambiados entre sus elementos. De este modo, ofrecen una visión general del funcionamiento del sistema en base a los mensajes intercambiados. Es de notar que en estos diagramas sólo se indicarán los mensajes intercambiados y no la información que los acompaña con propósitos de claridad. Si se quiere consultar en detalle la información que acompaña a cada tipo de mensaje, se deben consultar los epígrafes 3.4.4.2 y 3.4.4.3.

Entrada en partida individual

En la Figura 3.8 se puede ver un caso simplificado de los mensajes intercambiados entre cliente y servidor cuando un usuario ya existente se identifica para entrar a una partida individual.

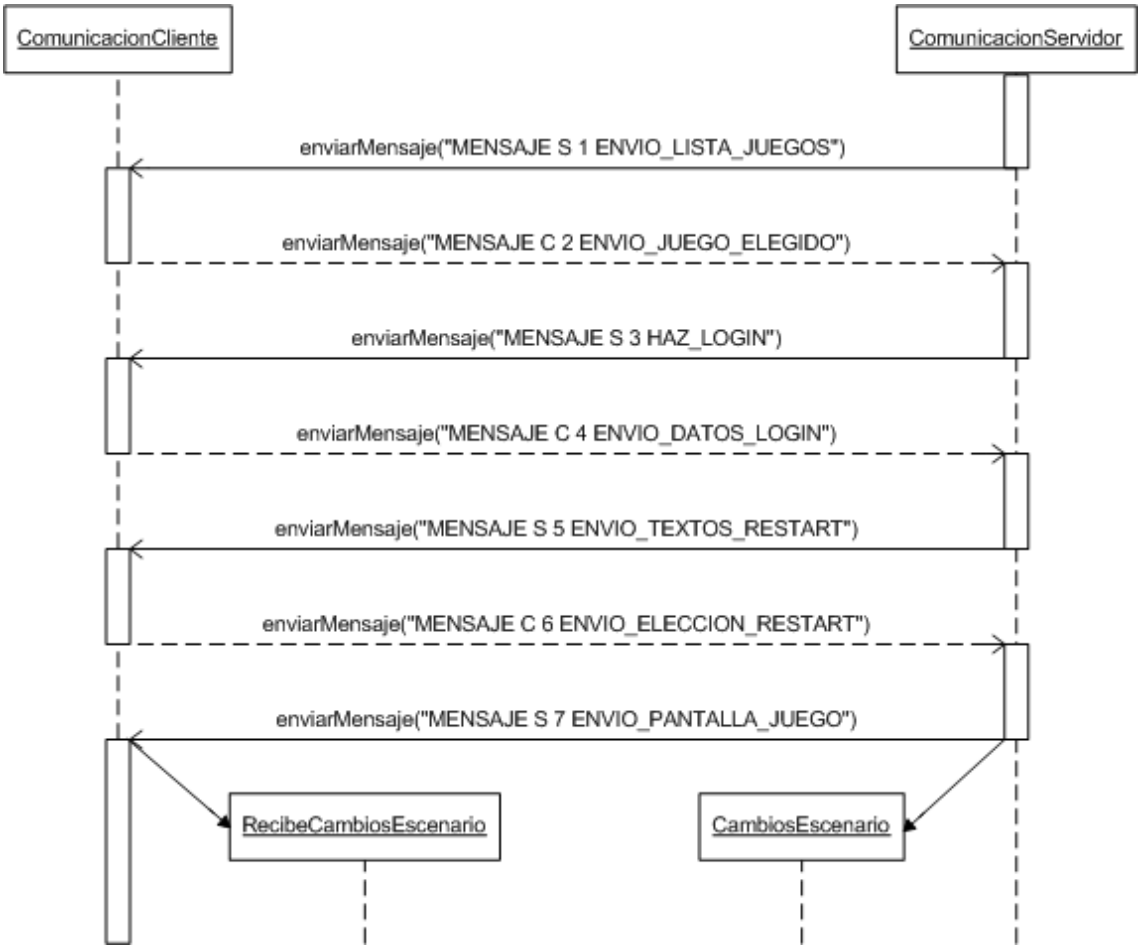


Figura 3.8 Entrada en partida individual

Como se puede observar, los dos extremos que implementan el protocolo en el cliente y servidor son, respectivamente, *ComunicacionCliente* y *ComunicacionServidor*. Un rectángulo estrecho indica que el objeto está activo, en el caso del cliente esto se referirá a

que el usuario puede interactuar con el móvil, mientras que en el caso del servidor se referirá a que está realizando un procesado.

El ejemplo anterior ilustra los mensajes intercambiados entre *ComunicacionCliente* y *ComunicacionServidor* en el proceso de identificación de un usuario ya existente que quiere entrar en una partida individual. Así, el usuario entra a la aplicación y se le envía la lista de videojuegos de la plataforma. Tras elegir uno, se le envía la información necesaria para identificarse. El usuario introduce sus datos y los envía. Al ser un usuario ya existente que quiere entrar en una partida individual, se le presenta la opción de reanudar el juego por donde lo dejó o bien comenzar desde el principio. El usuario elige una de las dos opciones y la envía al servidor, que le envía ya la primera localización de juego. En este punto, con el juego ya iniciado, se lanza *CambiosEscenario* por parte del servidor y *RecibeCambiosEscenario* por parte del cliente, que estarán preparados para intercambiar las peticiones de actualización del contexto de juego que sean necesarias.

Registro en partida individual, entrada o registro en partida multijugador

En la Figura 3.9 se puede ver un ejemplo simplificado de los mensajes que se intercambian entre cliente y servidor en el proceso de registro de usuario en una partida individual, o en el proceso de entrada o registro en una partida multijugador.

Figura 3.9 Registro en partida individual, entrada o registro en partida multijugador

Se puede apreciar que el proceso es muy parecido al de la Figura 3.8, siendo la única diferencia que no se pregunta al usuario si desea continuar con sus progresos o reiniciar la partida. Esto ocurre porque si se trata de un registro nuevo, no tendría sentido reiniciar los progresos, al estar ya reiniciados. Y si se trata de una identificación en partida multijugador sólo se da opción a recuperar los progresos guardados, ya que si no, un único usuario podría reiniciar la partida a la que están jugando otros y destruir sus progresos.

Bucle normal de juego

En la Figura 3.10 se puede ver un ejemplo del bucle normal de juego, que consiste en el intercambio entre cliente y servidor únicamente de los dos tipos de mensaje mostrados. El servidor envía todos los datos relativos a la localización a mostrar al usuario (imágenes, textos, objetos, etc.), y el usuario, por su parte, selecciona la acción que quiere realizar y se la envía al servidor junto con otros datos. Esto provoca un envío de los datos relativos a la nueva localización del servidor al cliente, con lo que se continúa el bucle.

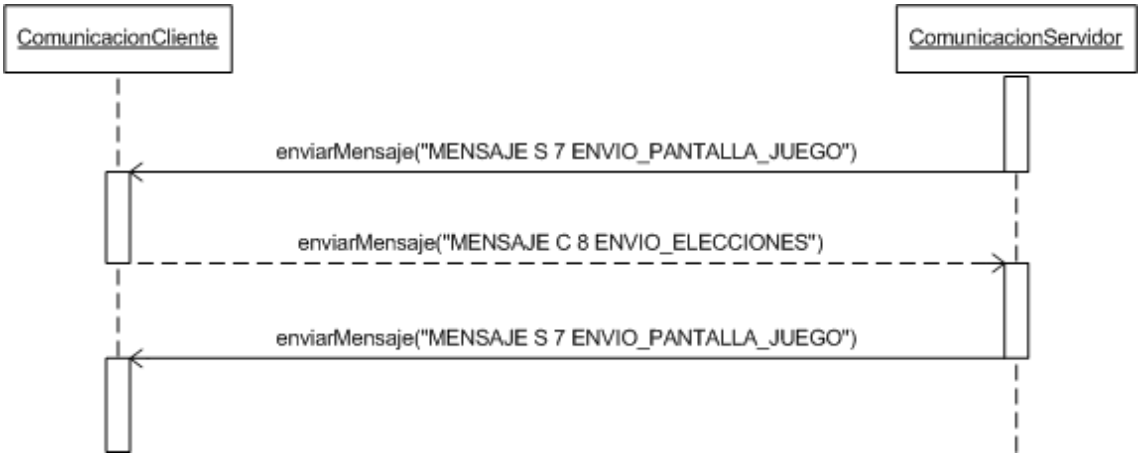


Figura 3.10 Bucle normal de juego

Usuario que provoca cambios en el modelo o entorno de juego

En la Figura 3.11 se puede ver el proceso por el cual un usuario de una partida multijugador agrega cambios realizados al modelo o al entorno de juego, para que el resto de usuarios comprueben si dichos cambios les afectan y, en su caso, realicen una recarga del contexto de juego. Este proceso también ocurre cuando un usuario entra a una localización que tiene definido un tiempo límite, lo que puede ocurrir tanto en partidas individuales como colectivas.

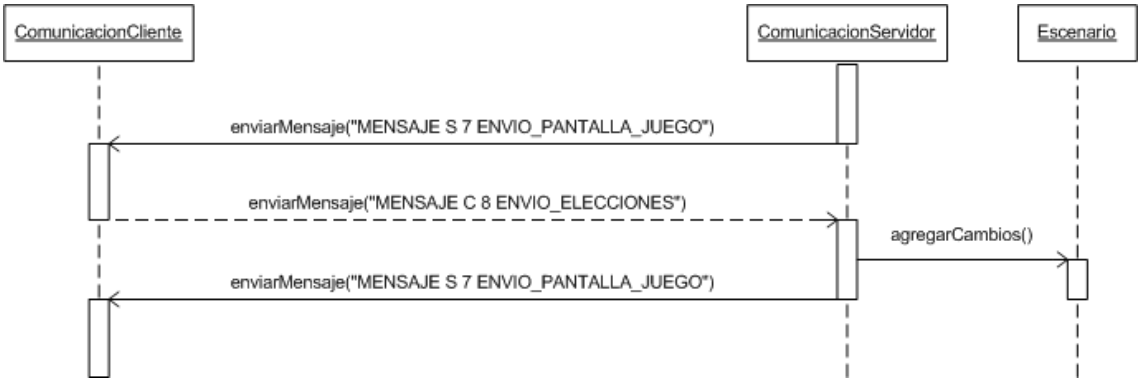


Figura 3.11 Usuario que provoca cambios en el modelo o entorno de juego

Como se puede apreciar, el proceso es similar al de la Figura 3.10, la única diferencia es que el usuario sabe que está en una partida multijugador, por lo que tiene que avisar de que ha realizado cambios a los archivos XML que conforman el modelo de la aplicación o al entorno de juego (por ejemplo, al entrar o salir de una localización). Este mecanismo también ocurre en el caso de que se entre en una localización que tiene definido un temporizador, en cuyo caso se extrae la duración del límite de tiempo de los archivos XML y se lanza un temporizador de esa duración. Cuando dicho temporizador expire, el servidor llamará a *agregarCambios()* de forma asíncrona, para forzar así una actualización del cliente.

Actualización del contexto de juego

En la Figura 3.12 se puede apreciar el intercambio de mensajes que ocurre cuando se han realizado cambios al modelo o al entorno de juego que afectan a un determinado usuario. En el diagrama se puede ver el proceso que sigue este usuario para recargar su contexto de juego.

El servidor es el único que conoce el estado de todos los jugadores en todo momento, por eso es él quien debe encargarse de notificar a los clientes de que tienen que hacer una petición de recarga de pantalla. Esto lo hace por medio de los hilos *CambiosEscenario* y *RecibeCambiosEscenario*, que están a la espera de que haya cambios en el modelo o en el entorno de juego.

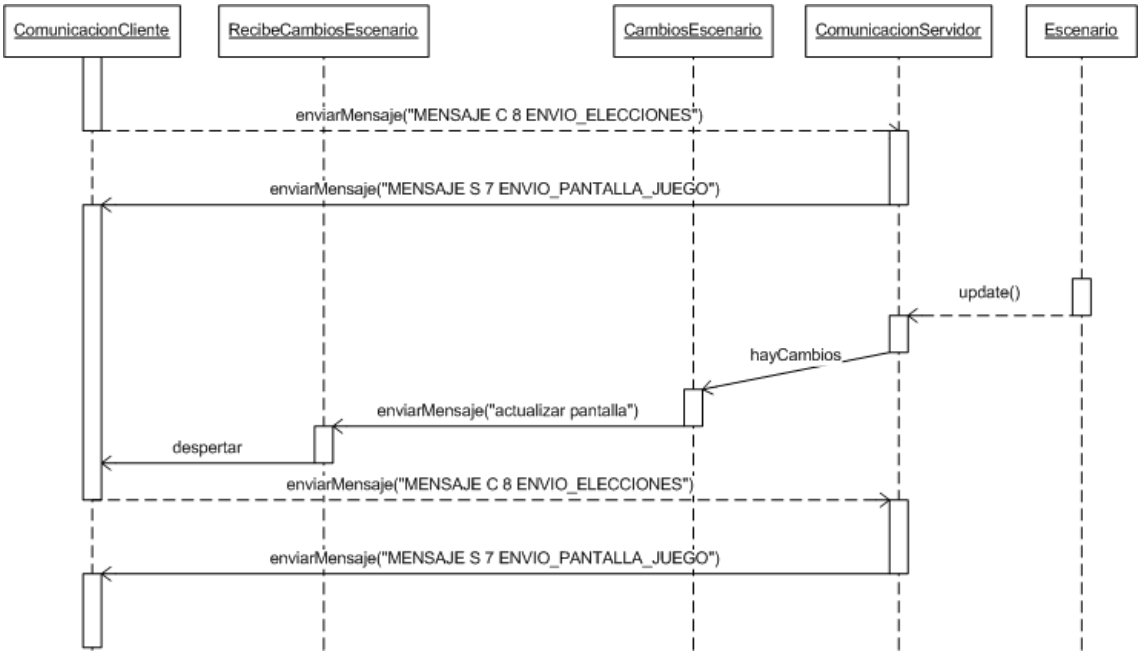


Figura 3.12 Actualización del contexto de juego

Como se ve en el diagrama, el jugador está realizando un proceso de juego normal, en el que selecciona una acción y el servidor le envía el nuevo contexto de juego producido como consecuencia de dicha acción. Tras esto, se puede ver que alguien ha introducido cambios en *Escenario*, que es el sujeto del patrón observable y, por tanto, notificará de estos cambios a todos los usuarios que se encuentren en esa pantalla. Para ello, se llama al método *update()* de *ComunicacionServidor*, que comprueba si los cambios ocurridos son relevantes al usuario.

Llegado ese punto, y si los cambios son relevantes, se notifica a *CambiosEscenario*. Dependiendo del tipo de cambio realizado, éste escoge uno de los mensajes de su vocabulario (en el ejemplo propuesto, "actualizar pantalla") y se lo envía a *RecibeCambiosEscenario*, que lo procesará y señalizará al cliente para que éste realice una petición de actualización de la pantalla en la que está. Para ello, se realiza el mismo proceso que para el envío normal de elecciones, por lo que en primer lugar se envía la variable *seguirJugando*, que en este caso valdrá "ok" al no haber elegido abandonar la aplicación ni recomenzar el videojuego. A continuación se envían las variables *accion*, *opcion* y *usarcon* vacías, esto es, sin seleccionar ninguna acción, por lo que el servidor sabe que al estar estas variables vacías y tener que *seguirJugando* es "ok", entonces lo que el cliente está pidiendo es, simplemente, una actualización de la pantalla en la que está con los nuevos datos ya modificados del modelo.

Tras esto, el servidor envía la localización actualizada al usuario, y éste puede continuar con el juego sin más interrupciones. Hay que notar que esto ocurre simultáneamente para todos los jugadores que se encuentren en la misma pantalla en la que se han provocado cambios, y que el resto de jugadores que se encuentren en otras localizaciones de la misma partida, o directamente en otra partida, no recibirán notificación alguna para realizar la recarga de sus datos del modelo. De este modo se garantiza que no habrá interrupciones al resto de usuarios a menos que sean completamente necesarias para garantizar que no hay problemas con la coherencia de los datos y que se mantiene actualizada la lista de jugadores con los que se comparte pantalla, con los que se puede interactuar (hablando con ellos, o intercambiando objetos, por ejemplo).

Abandono de partida

En la Figura 3.13 se puede ver el intercambio de mensajes que ocurre cuando un usuario decide abandonar la partida en la que se encuentra.

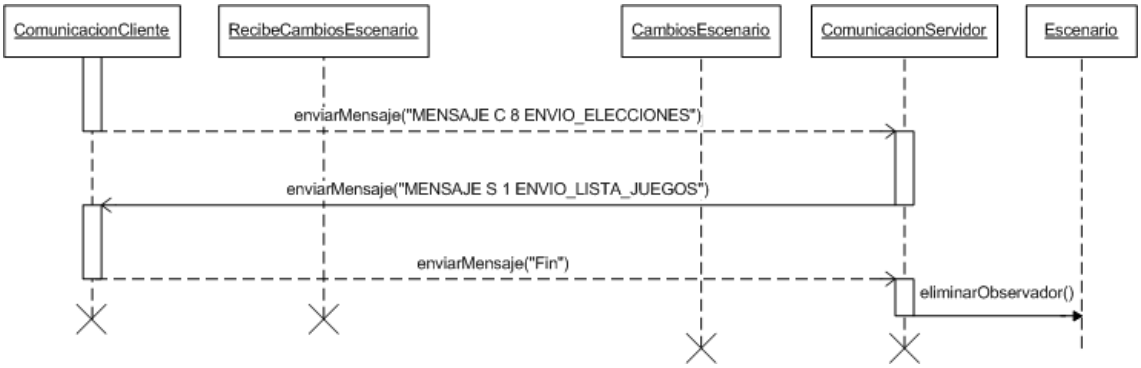


Figura 3.13 Abandono de partida

En el diagrama se muestra el proceso de abandono de partida considerando que el usuario se encuentra en el bucle de juego cuando decide marcharse. Desde el cliente se realiza el envío de la variable *seguirJugando*, que debe contener el texto “salir” al haber elegido el usuario abandonar la partida. Por tanto, el servidor lo recibe y contesta enviando al usuario la lista de videojuegos de la plataforma en el caso de que quisiera jugar a otro de los juegos. Desde la lista, el usuario tiene una opción de salir de la aplicación que envía el mensaje “Fin” al servidor, tras el cual se eliminan los hilos del cliente y los del servidor. En este último caso, si se estaba en una partida multijugador, el servidor se encarga de eliminar al usuario de la estructura que maneja a los usuarios que están jugando en ese momento, y hace que *Escenario* borre a este usuario como observador.

Si el usuario se encontraba en la presentación del videojuego, también tiene una opción para abandonarlo que hace el mismo envío del mensaje “Fin” al servidor ya descrito. Por último, si el usuario se encontraba en la pantalla de bienvenida de la aplicación, puede abandonarla sin necesidad de enviar ningún mensaje, ya que en ese momento aún no se ha establecido una conexión de red.

Fin de partida

En la Figura 3.14 se puede ver el diagrama con los mensajes intercambiados cuando un usuario finaliza con éxito un videojuego de la plataforma.

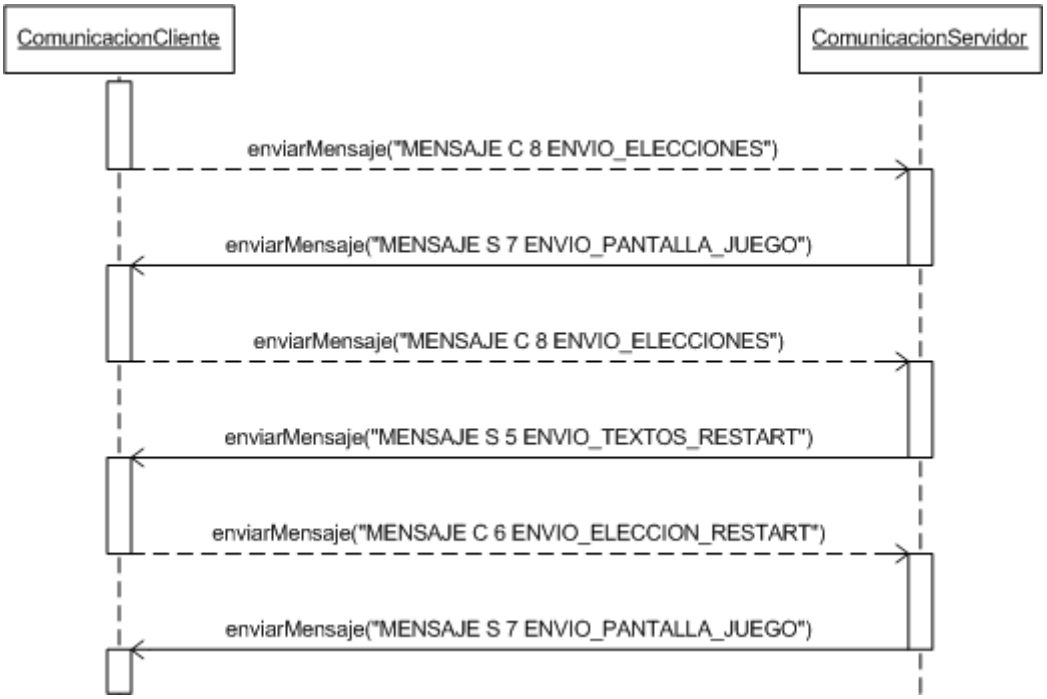


Figura 3.14 Fin de partida individual

En primer lugar se puede ver que el usuario envía al servidor la acción elegida, que en este ejemplo va a ser la acción que hace finalizar con éxito el juego. Tras esto, el servidor contesta con un mensaje normal de envío de pantalla de juego. La diferencia estriba en la localización que se envía al usuario, que en este caso es una localización especial en la que se muestra un texto de felicitación al usuario, y se da opción de salir o volver a comenzar el juego. En el ejemplo el jugador elige volver a comenzar, por lo que realiza un envío de *seguirJugando* que ha de ser igual a “recomenzar”. Al recibir ésto el servidor y tratarse de un usuario individual, se procede con el envío de los formularios necesarios para que el usuario elija si desea reiniciar la partida. Si elige que no, se le vuelve a mostrar el texto de felicitación. Si elige que sí, se notificará al servidor que reiniciará los progresos de la

partida y enviará al usuario el contexto de juego correspondiente a una partida recién creada.

Hay que reseñar que, en el caso de una partida multijugador, no se enviarían los dos mensajes correspondientes al reinicio de la partida. Además se añadiría un cambio al sujeto observable *Escenario* cuando el usuario realizó la última acción que le hizo finalizar la aventura, para que el resto de usuarios de la partida realizaran una actualización de su contexto de juego y poder así mostrarles que un usuario ya había finalizado la aventura con éxito.

CAPÍTULO 4: MANUAL DE USUARIO

En este capítulo se esbozará un manual de usuario en el que se expondrán las consideraciones a tener en cuenta tanto desde el punto de vista de un usuario administrador del sistema, como de un usuario jugador. En el primer caso, se comentarán los pasos a seguir para desarrollar e incorporar nuevos videojuegos a la plataforma, así como el proceso para poner en marcha el servicio. En cuanto al segundo caso, estará enfocado al uso del sistema por parte de un jugador, y se explicará la navegación y el uso de las opciones que ofrece la plataforma.

4.1 MANUAL DE USUARIO ADMINISTRADOR

El objetivo de este apartado es proporcionar una guía básica a un usuario administrador sobre la confección de nuevos videojuegos para la aplicación, así como los pasos necesarios para poner en marcha el servidor y así poder proporcionar el servicio de videojuegos desarrollado a los usuarios. Esta guía servirá de manual de referencia en el que se definen los pasos y acciones a llevar a cabo para confeccionar juegos e incorporarlos a la plataforma de forma ordenada y clara.

4.1.1 ESPECIFICACIÓN DEL VIDEOJUEGO

El primer paso a contemplar es la confección de la idea del videojuego. Se recomienda realizar un boceto de la historia en el que se defina a grandes rasgos la situación en la que se desarrollará el argumento del videojuego, las diferentes localizaciones en las que se dividirá la aventura, los objetos que podrían ser necesarios, las acciones que harán avanzar al jugador, etc.

Esto es indispensable a la hora de realizar un desarrollo claro y ordenado de un videojuego para esta aplicación, dada la variedad de archivos que entran en juego. Igualmente es recomendable tener conocimientos básicos del lenguaje XML, aunque no es un requisito imprescindible, dada la sencillez de los archivos empleados y lo evidente de la organización del sistema de etiquetas en árbol.

4.1.2 ESTRUCTURA Y CONTENIDOS

Para cada videojuego a desarrollar para la plataforma, dentro del directorio que lleva el nombre que lo identificará en la lista de juegos, se ha de crear una estructura de directorios como la mostrada en la Figura 4.1. Se deben crear esos tres directorios con esos nombres (teniendo cuidado con las mayúsculas y minúsculas). En el directorio "files" se crearán los archivos XML que configuran el videojuego. En el directorio "images" se introducirán las imágenes necesarias a lo largo de toda la aventura, y finalmente en el directorio "XSD" se introducirán los archivos *Schema* (*.xsd*) que validen los archivos de configuración creados, para de esa forma asegurar que no ha habido ningún error semántico a la hora de crearlos.

Una vez creada la estructura de directorios, solo falta crear cada uno de los archivos necesarios para especificar las localizaciones, textos, nombre de las imágenes y cómo se reparten en el juego, objetos a usar, acciones, etc. Para lo que habrá que tener en cuenta que el servidor debe ser capaz no solo de leer y servir dichos archivos a los clientes, sino de escribir y modificar parte de sus contenidos, por lo que se deben tener activos los permisos de lectura y escritura en el directorio creado.

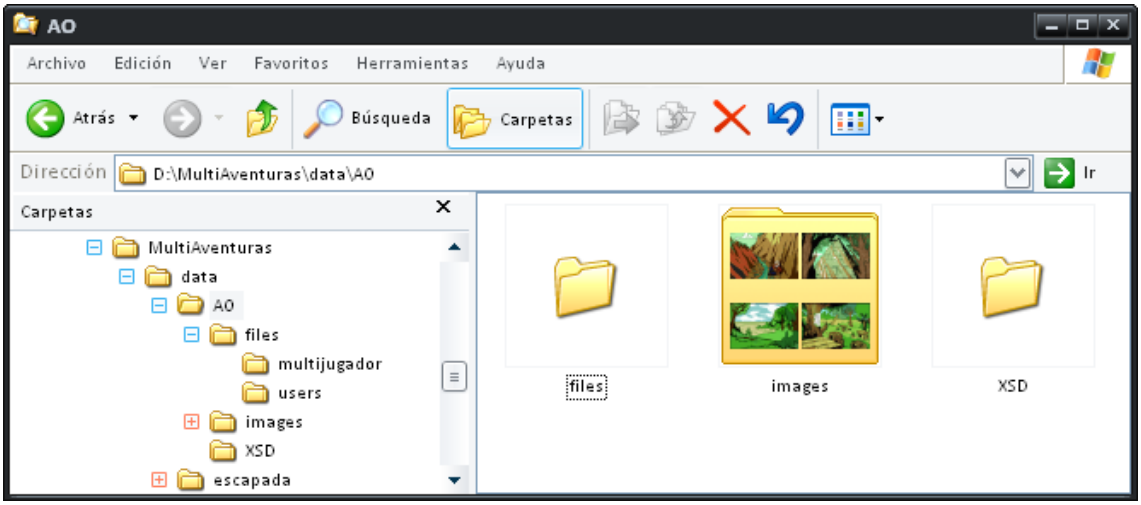


Figura 4.1 Estructura de directorios de cada videojuego

Cabe reseñar que para el proceso de creación de nuevos videojuegos para la plataforma que se describe a continuación, se recomienda previamente la lectura del apartado 3.3 sobre los archivos de configuración del motor desarrollado en [15], así como la lectura del apartado 3.4.1.2 de este documento, en el que se detallan las modificaciones realizadas a dichos archivos.

Para la creación de los archivos de configuración de un nuevo videojuego, se recomienda seguir un proceso similar al siguiente:

1. Confeccionar un mapa de forma esquemática que contenga todas las localizaciones o escenas que se le podrán presentar al usuario, indicando para cada una de ellas el nombre de la imagen que se mostrará para ilustrarla, el texto descriptivo de esa localización y, sobre todo, las posibles direcciones que podrá tomar el usuario, indicando a que localización le llevarán. Se recomienda realizar este esquema dibujándolo sobre papel o haciendo uso de un entorno gráfico que posibilite el entendimiento claro y total del mapa de la historia.

2. Una vez se tenga ese mapa de forma esquemática, se hace necesario pasar dicha información a un formato que pueda entender la aplicación. Esto se realiza creando o modificando el archivo *map.xml* perteneciente al directorio “files” del videojuego a desarrollar. En ese archivo se creará un elemento con el nombre de cada una de las localizaciones que participan en la aplicación: tanto las iniciales “<screenIndex>”,”<screenLogin>”, etc. como las específicas para cada una de las localizaciones en las que se podrá mover el usuario dentro de la historia planteada, recogidas bajo entradas de tipo “<screen id=”x” version=”y” timed=”z”>”. Para cada una de estas últimas, se incluirá el texto descriptivo que acompaña a la imagen, la URL de la imagen que se desee mostrar (que estará contenida en el directorio “images”), y las direcciones posibles a tomar desde esa localización, usando para ello el elemento “<direction>” y los sub-elementos “<d id=”x” activ=”si/no”> texto </d>” que especifican cada una de las direcciones. Si fuera necesario definir una versión alternativa de alguna localización, porque pueda cambiar tras las acciones del usuario, esto se hará bajo un epígrafe “<alt>” que contendrá, a su vez, la estructura de la nueva localización que se tendrá que mostrar tras haber ocurrido los cambios. Asimismo, se indicará en el atributo “version” de “<screen>” la versión actual de la localización.

3. El siguiente paso corresponde a la definición de los diferentes objetos de la aventura. Al igual que en el punto anterior, se recomienda colocar esos objetos sobre el mapa esquemático del videojuego, especificando dónde puede encontrar el usuario dicho objeto inicialmente, y adicionalmente algunas características del mismo, como puede ser si puede cogerse o no. Asimismo, en este punto será

también donde se defina el lugar donde se van a situar los NPC con los que el jugador interactuará a lo largo de la historia.

4. Para hacer que los objetos y NPC sean accesibles desde la aplicación, es necesario transferir el esquema realizado a formato electrónico, creando para ello el archivo *object.xml*, en el que no solo se definen los objetos y su situación inicial, sino también una serie de atributos de los mismos que se deben especificar de la forma más detallada posible. Se recomienda con el fin de organizar los contenidos de este archivo no definir los objetos que intervienen en la historia siguiendo el orden en que serán usados (ya que eso depende de las acciones que lleve a cabo el usuario y del orden en el que lo haga, por lo que siempre existirá una fuente de incertidumbre), sino organizando los objetos según la localización del juego en que aparecen por primera vez.
   No obstante, eso no varía el desarrollo del juego, tan solo facilita la posibilidad de modificar dichos objetos en caso de que sea necesario. Así, para cada objeto a crear en la historia se deben especificar todos y cada uno de los atributos de los que consta el elemento "<ob id="u" screen="v" vers="w" cogible="x" usable="y" npc="z">, como ya se vio en detalle en el apartado 3.4.1.2.
   Adicionalmente, se podrán añadir una serie de textos descriptivos de las diferentes acciones que se pueden realizar sobre dicho objeto, así como una descripción propia del objeto que se muestre al usuario al llegar a la localización del videojuego en la que se encuentra el objeto tratado.

5. Para acabar con la creación de archivos de configuración, se deben especificar las diferentes acciones a llevar a cabo en la aventura para poder avanzar, contenidas en el archivo *actions.xml*. Se pueden especificar todas las acciones que se deseen, y se recomienda agruparlas, de nuevo, por la localización en la que se hace necesario su uso, facilitando de esa forma una estructuración que permita una comprensión lógica de las acciones a lo largo de la historia. Se puede pensar que ordenarlas de forma cronológica sería la mejor opción, pero dado que en este tipo de aventuras la cronología de las acciones no es fija, sino que depende de las acciones y preferencias del usuario, ésta no es una opción recomendable.
   También hay que reseñar que es en este archivo donde se define el guión que van a seguir las conversaciones con los NPC de la aventura. Se recomienda dedicar un tiempo a la escritura de un buen diálogo que contenga múltiples ramas de

conversación que se vayan abriendo y por las que pueda ir avanzando el usuario (incluidas algunas que no lleven a nada), ya que un buen guión será una de las mayores virtudes que pueda tener un juego de este tipo para mantener enganchado al usuario a la historia. Para ello, se usa un elemento de tipo <hablarcon npc="x">, como se vio en detalle en el apartado 3.4.1.2.

Inicialmente puede servir de ayuda realizar una descripción de cómo podría resolverse toda la aventura, a forma de posible solución del creador. De esta forma se previene el olvido de algún objeto o acción necesaria en el transcurso de la historia. Una vez hecho esto, se separan los objetos y las acciones en sus correspondientes archivos, y se organizan en función de las localizaciones en las que se hacen necesarios.

Se recomienda especial precaución a la hora de definir una nueva acción, para especificar de forma correcta el objeto u objetos sobre los que se realizará, teniendo en cuenta en todo momento que los nombres de los objetos deben haber sido definidos con anterioridad en el archivo *object.xml*.

6. Seguramente en la definición de las acciones se haga necesario modificar objetos que ya habían sido definidos (alterando alguno de sus atributos o redefiniendo alguno de sus textos), o incluso crear objetos que no se tuvieron en cuenta en el paso 3. Es por eso que el desarrollo de un videojuego no es tan sencillo ni secuencial, sino que se hace necesario seguir varias veces estos pasos de forma iterativa, hasta que el videojuego quede completamente confeccionado.

7. Una vez se ha completado la creación de los archivos implicados en la definición del videojuego, y antes de probarlo, es recomendable revisar la estructura y la correcta sintaxis de estos documentos. Para ello, se puede hacer uso de un validador de documentos XML para generar documentos que restrinjan su arquitectura, como son los archivos *Schema* (*.xsd*).

4.1.3 PUESTA EN MARCHA DEL SERVIDOR

Una vez se tengan desarrollados los videojuegos que se quieran poner a disposición del público, el paso siguiente es la puesta en marcha de un servidor que ofrezca este servicio. El servidor se proporciona en forma de un archivo *JuegoServidor.jar*, que tendrá que ir acompañado de un directorio “data” que contendrá en su interior los directorios de los

videojuegos disponibles en la plataforma, formando la estructura en árbol mostrada en la Figura 4.2:

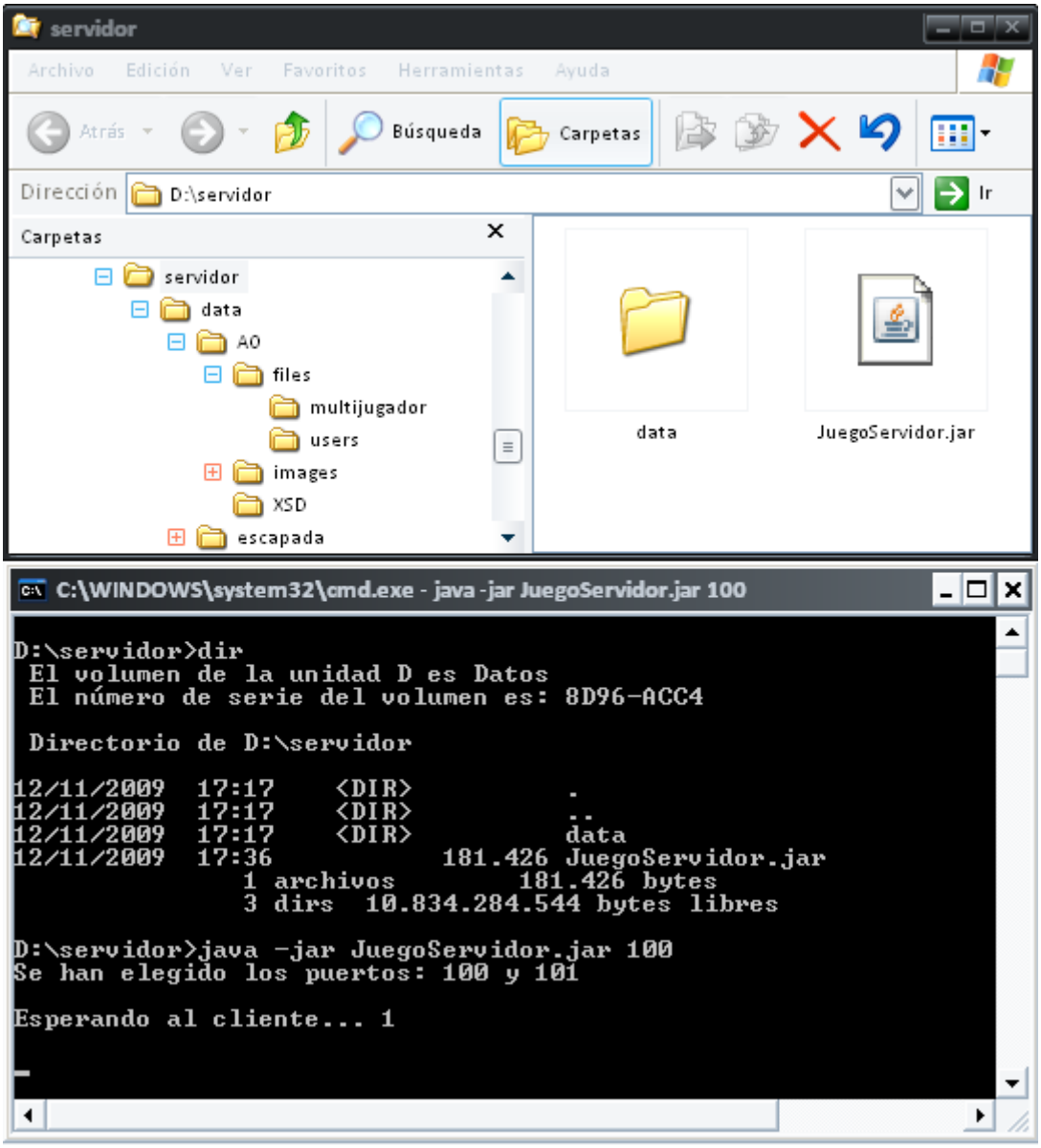


Figura 4.2 Estructura de directorios y llamada al servidor

Una vez dispuesta esa estructura de directorios, lo siguiente es hacer la llamada para lanzar el servidor. Para ello, la plataforma Java [30] ha de estar instalada en el sistema. La llamada al servidor será de la forma "java –jar JuegoServidor.jar puerto", como se puede ver en la Figura 4.2. Esta llamada es sensible a mayúsculas y minúsculas, por lo que habrá que ser cuidadosos para no cometer un error, además de no olvidar incluir la extensión *.jar* en ella. En "puerto" se indica el primero de los dos puertos consecutivos en los que va a escuchar el servidor, de modo que si, por ejemplo, se quiere poner el servidor a escuchar en los puertos 100 y 101, se tendría que realizar la llamada que aparece en la Figura 4.2. Si

no se especifica ningún puerto en la llamada al servidor, se tomarán, por defecto, los puertos 1234 y 1235.

Puede ocurrir que el puerto seleccionado esté ocupado por otra aplicación, por ejemplo si se elije el puerto 80 para escuchar a los clientes, y a la vez se tiene configurado un servidor web en ese mismo puerto. En ese caso, al lanzar el servidor se informa de que dicho puerto está ocupado, y se pide al usuario que seleccione otro, o bien que lo libere para que se pueda utilizar por el sistema.

Tras todo lo anterior, si no ha habido más errores, el servidor muestra la línea "Esperando al cliente… 1", con la que da a entender que ha comenzado la escucha en los puertos configurados y está esperando a que lleguen usuarios.

4.2 MANUAL DE USUARIO JUGADOR

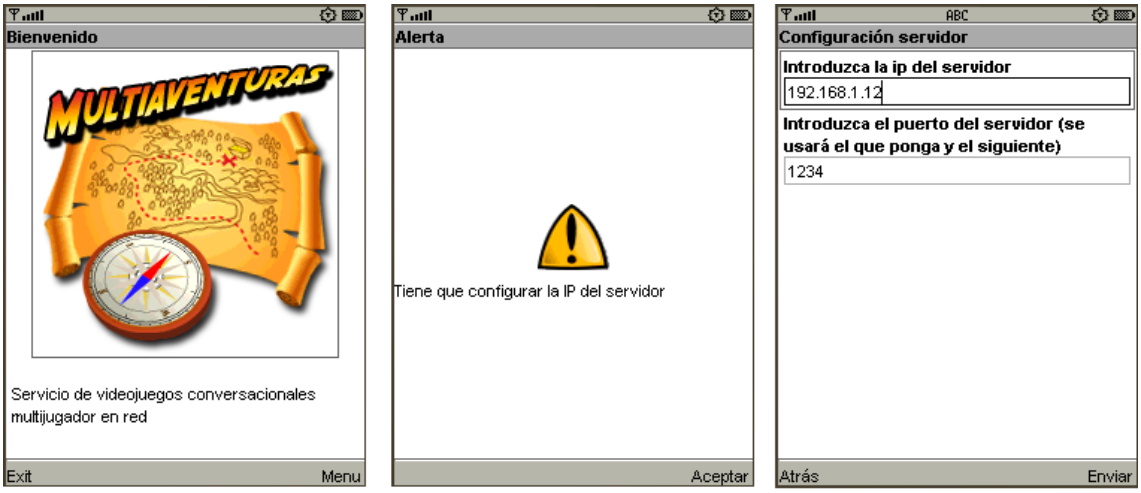
En este apartado se explicarán los pasos a seguir por un usuario para instalar y configurar la aplicación en su móvil, y se hará una somera descripción de la forma de jugar a uno de los videojuegos que ofrece la plataforma.

El usuario dispone de dos formas distintas de instalar la aplicación en su móvil:

- Descargando la aplicación, compuesta por un par de archivos *.jar* y *.jad*, desde Internet con un ordenador, tras lo cual tendrá que transferir la aplicación al teléfono móvil. Para ello se pueden usar distintos medios, desde la transferencia directa de los archivos *.jar* y *.jad* por Bluetooth, Wi-Fi o cable USB; hasta el uso de otro software que facilite la tarea de instalación, como por ejemplo Nokia PC Suite [31] para el caso de los móviles Nokia, o incluso el propio IDE NetBeans, que permite realizar este proceso por medio de su opción "*Deploy*".
- Realizando la descarga de la aplicación directamente desde el móvil. Este método se conoce como OTA (*Over The Air*), y consiste en conectarse a Internet desde el móvil, ya sea por Wi-Fi, WAP, GPRS, etc.; y acceder a la web en la que se encuentre la aplicación, que se instalará directamente en el móvil al seleccionarla.

Una vez instalada la aplicación en el móvil, hay que ejecutarla y configurarla. Para ello, primero será necesario ir al menú de aplicaciones del teléfono, desde el que tendrá que aparecer la opción para lanzar esta aplicación, que se mostrará bajo el nombre

"MultiAventuras". Al ejecutarla, se verá la pantalla de bienvenida que aparece en la Figura 4.3 (a). Al pulsar en "Menú", aparecerán dos opciones: "Entrar" y "Configurar". En el caso de que se haya jugado anteriormente, se podrá entrar directamente a la aplicación. Si no, será necesario configurar antes la dirección del servidor. Si se le da a entrar siendo la primera vez que se juega, el móvil informará de que antes tiene que configurarse la dirección del servidor, como se puede ver en la Figura 4.3 (b). La pantalla de configuración de la dirección del servidor, se muestra en la Figura 4.3 (c).



(a) (b) (c)

Figura 4.3 Entrada a la aplicación y configuración del servidor

En este punto, la aplicación se dispone a realizar la primera conexión con el servidor, por lo que se le pedirá permiso al usuario para hacerlo mediante un *prompt* del propio teléfono, como se ve en la Figura 4.4 (a). Si el usuario acepta que se establezca la conexión, se intentará, y si no hay problemas, se le mostrará la lista de videojuegos disponibles en la plataforma en ese momento, como muestra la Figura 4.4 (b). Una vez seleccionado uno de los videojuegos de la lista, se mostrará la pantalla de presentación de éste, que tendrá una apariencia similar a la que se ve en la Figura 4.4 (c).

Ya en esta pantalla, se puede elegir jugar en una partida individual o en una multijugador. En la Figura 4.5 (a) se puede ver el caso de que se elija una partida individual. En este caso se preguntará al usuario si desea crear un nuevo nombre de usuario, o identificarse para seguir jugando por donde quedó la última vez. Estas opciones se ven en la Figura 4.5 (b).

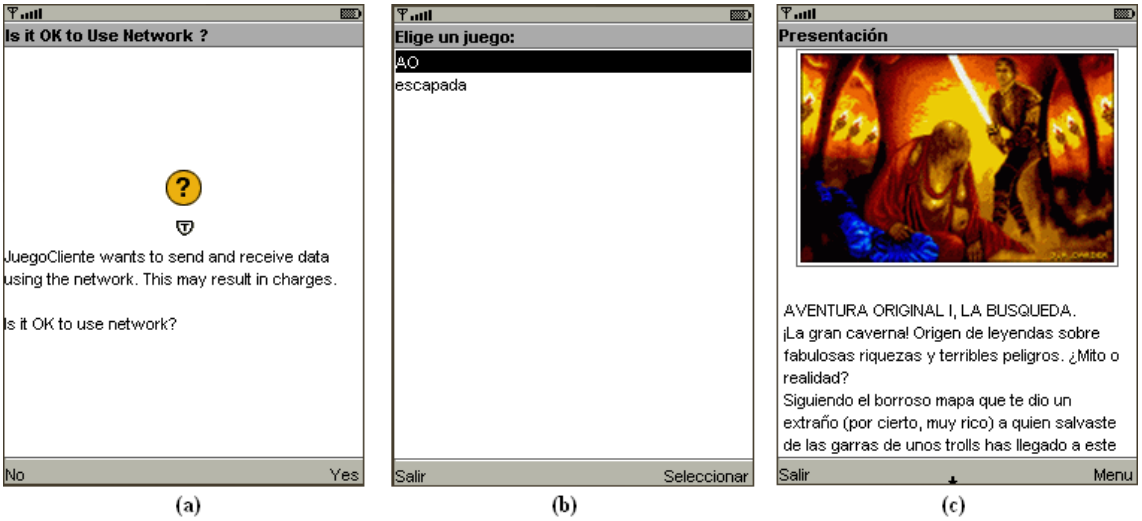


Figura 4.4 Selección de un juego y presentación de éste

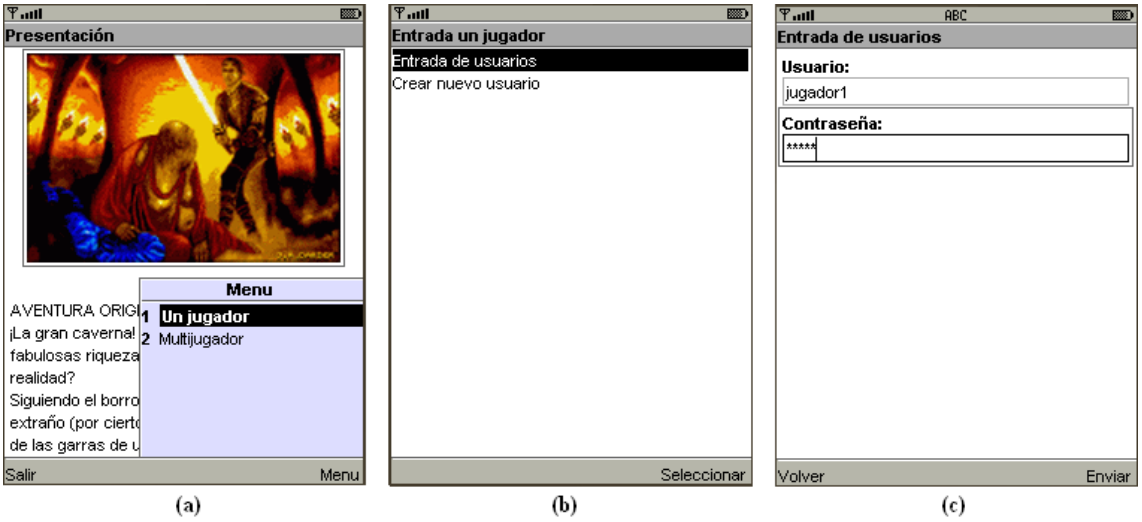


Figura 4.5 Entrada en partida individual

Si el usuario desea identificarse para entrar en una partida individual anteriormente creada tendrá que elegir la opción “Entrada de usuarios”, para a continuación introducir su nombre y contraseña como aparece en la Figura 4.5 (c).

Si, por el contrario, se elige jugar en una partida individual, pero registrando un nuevo nombre de usuario, habrá que elegir la opción “Crear nuevo usuario”, como muestra la Figura 4.6 (b). En este caso, al jugador se le presenta el formulario que se puede ver en la Figura 4.6 (c) en el que tendrá que introducir el nombre de usuario que quiere registrar así como una contraseña, que tendrá que ser introducida dos veces para asegurar que no ha habido errores al teclearla.

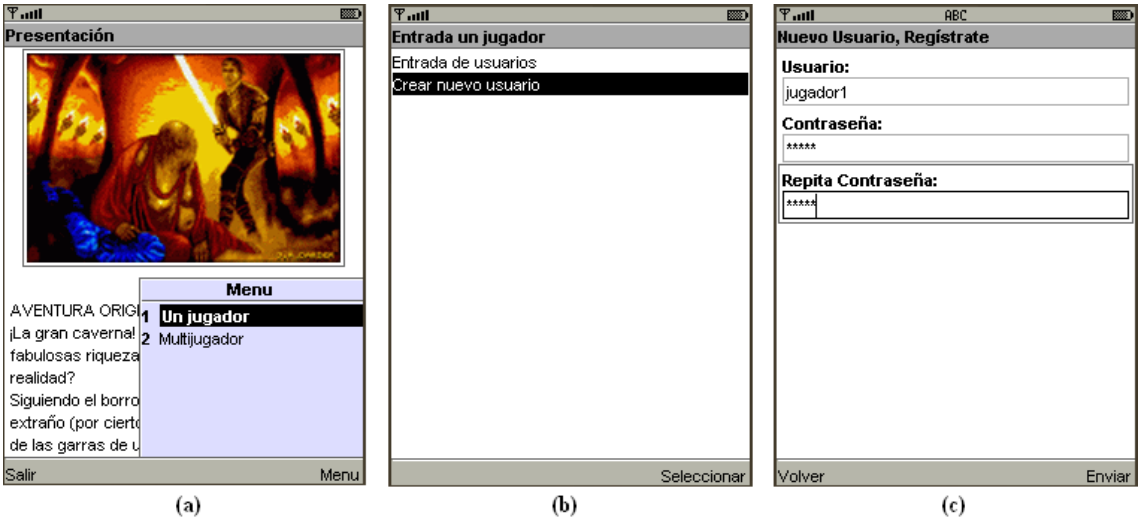


Figura 4.6 Creación de partida individual

Por el contrario, si se selecciona una partida multijugador como ocurre en la Figura 4.7 (a), se presentarán las opciones que se muestran en la Figura 4.7 (b), a saber: unirse a una partida multijugador ya existente, o bien la creación de una nueva partida multijugador. Si el usuario quiere entrar a formar parte de una nueva partida multijugador, seleccionará "Unirse a partida", tras lo cual se le presentará el formulario de entrada que se puede ver en la Figura 4.7 (c), donde se le requiere que introduzca el nombre de la partida, la contraseña establecida para ésta y su propio nombre de usuario.

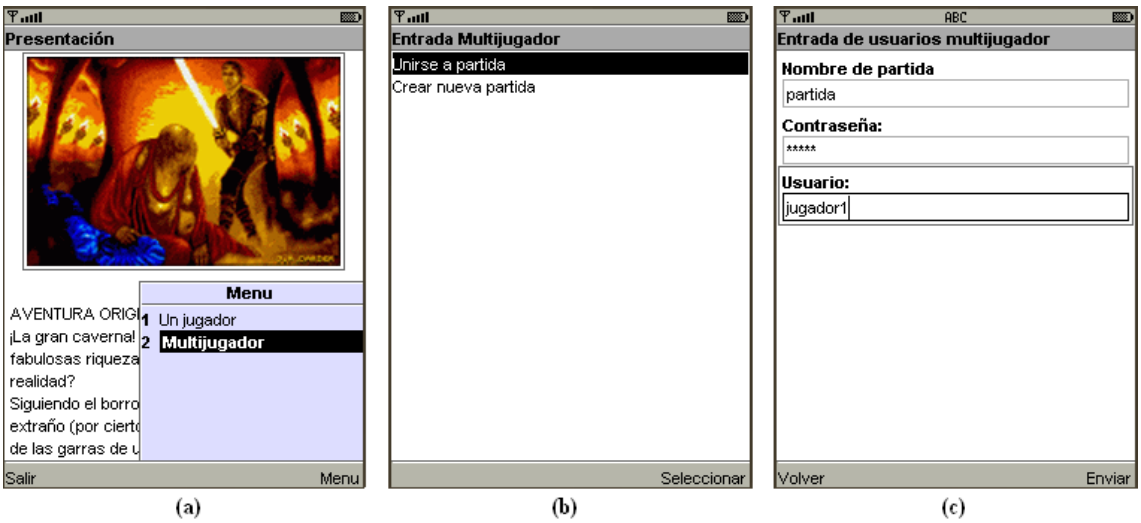
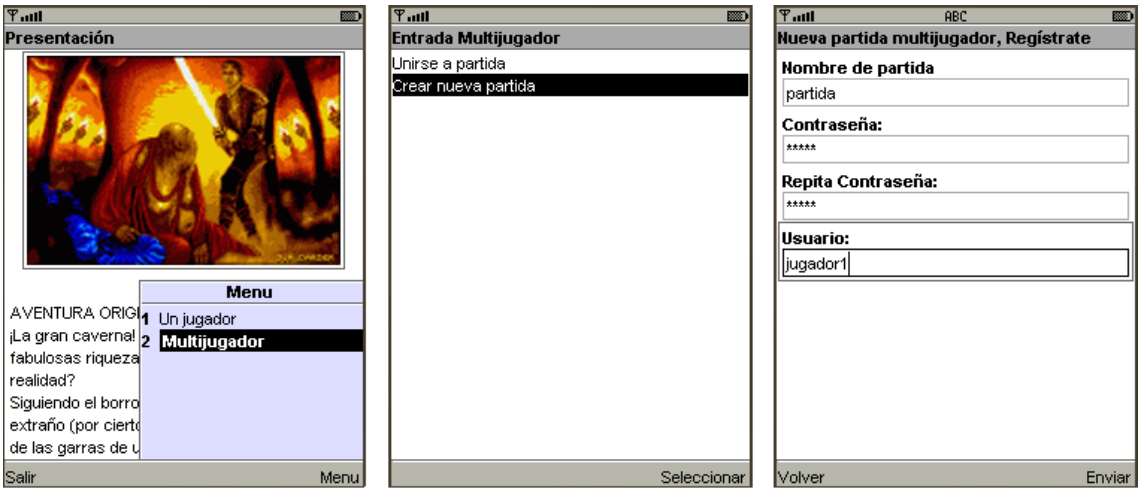


Figura 4.7 Entrada en partida multijugador

Por último, si el usuario desea crear una nueva partida multijugador, lo seleccionará como se muestra en la Figura 4.8 (b). Tras esto, se presentará el formulario de creación de partida multijugador que aparece en la Figura 4.8 (c), donde tendrá que introducir el

nombre que desea que tenga la partida, establecer la contraseña que han de conocer todos los usuarios que quieran participar (introduciéndola por duplicado para comprobar que no haya habido errores) y su propio nombre de usuario dentro de la partida.



(a) (b) (c)

Figura 4.8 Creación de partida multijugador

Durante estos procesos de identificación y registro se pueden producir diversos errores, entre ellos: que el usuario pretenda entrar en una partida que no existe, que intente crear una partida que ya existe (Figura 4.9 (a)), que intente unirse a una partida introduciendo una contraseña que no se corresponde con la definida para esa partida (Figura 4.9 (b)), que al crear una partida no coincidan los dos campos en los que se pide introducir la contraseña (Figura 4.9 (c)), etc. Si se produce alguno de estos errores, se avisará al usuario y se le permitirá intentar introducir la información de nuevo.

(a) (b) (c)

Figura 4.9 Ejemplos de errores en la identificación y registro

Si se ha seleccionado partida individual, y el usuario se identifica, se le preguntará si desea reiniciar la aventura desde el principio o continuar por donde dejó guardados sus progresos. Esto se puede ver en la Figura 4.10 (a). Comenzar de nuevo puede ser útil si el usuario se encuentra perdido y desea enfocar de nuevo el juego de otra manera.

En el caso de que se entre a una partida ya existente, puede ocurrir que el usuario sea la primera vez que lo hace, en cuyo caso comenzaría desde el principio, o ya haber jugado anteriormente en esa partida, caso en el que recuperaría sus progresos y podría continuar por donde iba.

Una vez realizado el registro o la identificación, tanto en caso de partida individual como multijugador, los datos se envían al servidor y durante un instante no se puede interactuar con el teléfono porque está realizando este envío de datos. Durante un momento se muestra la imagen que aparece en la Figura 4.10 (b). Al fin, una vez que se ha entrado en el juego, se muestra un escenario con su imagen definida, y su texto descriptivo que la acompaña, como se ve en la Figura 4.10 (c).

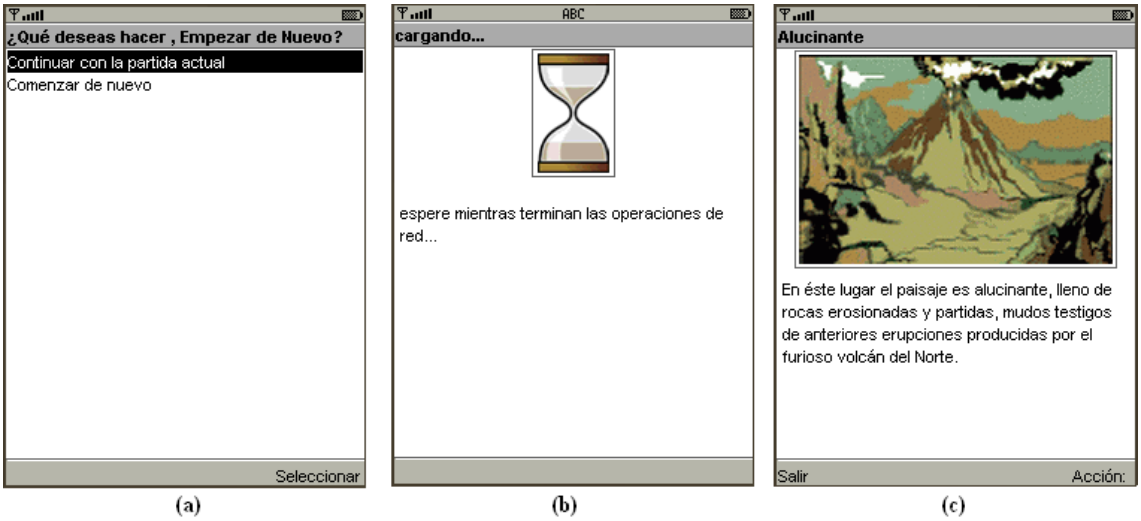


Figura 4.10 Entrada a un juego de la plataforma

Una vez en el juego, se pueden ver las opciones de “Salir” y “Acción”. Si el usuario quiere realizar cualquier acción, deberá pulsar el botón correspondiente en su móvil, y se le presentará una lista con las acciones que puede realizar, como se puede ver en la Figura 4.11 (a). Si selecciona por ejemplo la acción de moverse, se le mostrará a continuación una lista de direcciones a las que ir similar a la que se puede ver en la Figura 4.11 (b). Cuando

el jugador seleccione una de las direcciones, se moverá a esa localización si es posible, mostrándosele en tal caso el nuevo escenario en el que se encuentra.

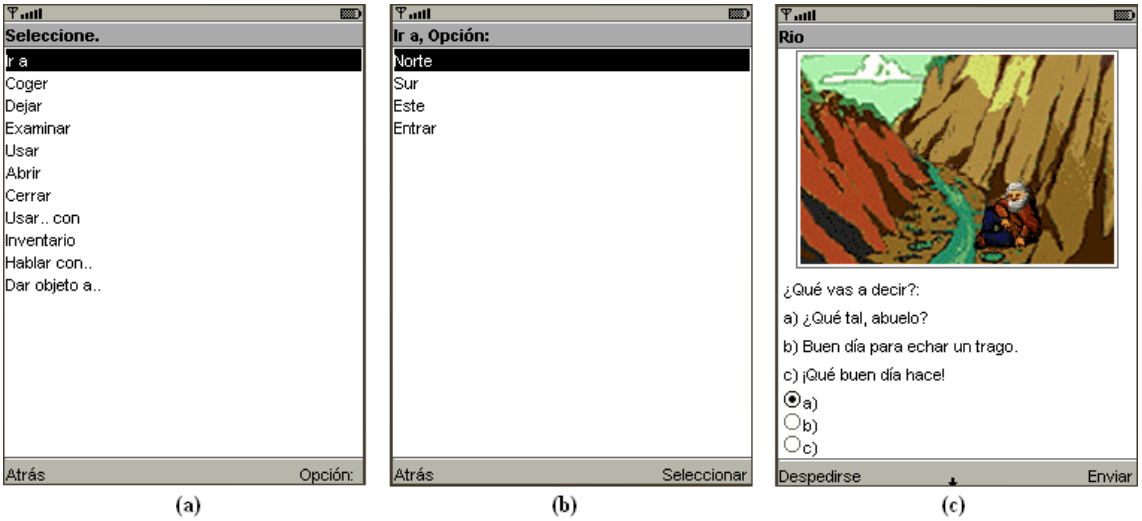


Figura 4.11 Diversas acciones en el juego

En la Figura 4.11 (c) se puede ver un ejemplo del aspecto que tiene un diálogo de los que se pueden establecer con un NPC. Se presentan una serie de posibles frases que el usuario puede decirle, el usuario escoge una y la envía, y acto seguido recibirá la respuesta correspondiente del NPC.

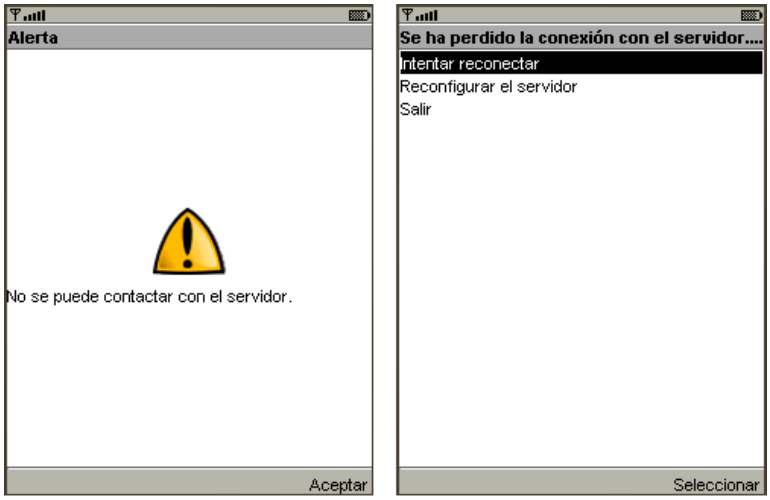
Cuando se está en una partida multijugador, se ofrecen nuevas opciones, como la de hablar con otro jugador, como se puede ver en la Figura 4.12 (a). Seleccionada la opción "hablar a", si hay otros jugadores en esa misma pantalla y se elige hablar con uno de ellos, aparecerá una caja de texto en la que el usuario puede escribir y mandarle el mensaje que desee al otro jugador. Si hay más de dos jugadores en esa misma localización, aparecerá la opción "hablar a todos", con la que se podrá enviar un mensaje de texto que se muestre simultáneamente a todos los jugadores que estén en dicha localización. En la Figura 4.12 (b) se puede ver la forma que tienen estos mensajes una vez que son recibidos. En esta misma imagen también se puede apreciar el efecto que tienen los jugadores en una partida multijugador sobre sus compañeros de escenario, aquí, por ejemplo, se muestra el texto "jugador3 llega", esto se debe a que anteriormente el usuario que está jugando se encontraba en ese escenario, y en un momento determinado, llegó a ese escenario el personaje "jugador3", por lo que es necesario informar al primero de su entrada, para que sepa que ahora puede, entre otras cosas, hablar con él o darle objetos.

(a) (b) (c)

Figura 4.12 Diversas acciones con efectos en otros jugadores

Por último, en la Figura 4.12 (c) se puede ver la alerta que se muestra a un usuario cuando realiza una acción, como en este caso es coger un objeto. Este objeto desaparecerá inmediatamente del escenario, ya que ya hay un usuario que lo ha recogido. Cuando un usuario que comparte escenario con otros en una partida multijugador provoca cambios en éste, por ejemplo cogiendo un objeto, se actualiza dicho escenario para el resto de usuarios, que verán momentáneamente una imagen similar a la ya vista anteriormente en 4.10 (b).

Si ocurre un error en la conexión con el servidor durante cualquier momento del proceso de juego, se mostrará en el teléfono el aviso que aparece en la Figura 4.13 (a). Tras esto, cuando el usuario haya leído el aviso, se le mostrará una lista con las tres opciones que aparecen en la Figura 4.13 (b): intentar reconectar, reconfigurar el servidor y salir. Si el usuario intenta reconectar, se intentará volver a establecer la conexión con el servidor, y si no es posible, se volverá a mostrar la lista con las tres opciones para que pueda seguir intentándolo. Reconfigurar el servidor no tiene sentido si se ha estado jugando correctamente y se ha interrumpido la conexión, aunque sí que lo tiene si no se puede conectar a la plataforma desde un primer momento, lo que podría estar causado por haber cometido algún error al introducir los datos de conexión del servidor. En ese caso, se permite introducir de nuevo la IP y el puerto del servidor. Por último, también se presenta la opción de abandonar el juego, en caso de que los problemas no se solucionen o que el usuario desee salir de la plataforma.



(a) (b)

Figura 4.13 Error en la conexión con el servidor

CAPÍTULO 5: PLAN DE PRUEBAS

En este capítulo se detallarán las pruebas realizadas a la aplicación desarrollada para verificar su correcto funcionamiento en la mayor cantidad de casos posible. Además de las pruebas de funcionamiento también se comentarán las medidas de tráfico realizadas, que serán relevantes de cara al uso de la aplicación en un contexto real.

5.1 MEDICIONES DEL TRÁFICO GENERADO

5.1.1 CONSIDERACIONES PREVIAS

La finalidad de la aplicación desarrollada es proporcionar un servicio de videojuegos conversacionales. Esta aplicación consta de dos partes, un servidor remoto y un *MIDlet* que tendrá que instalarse en los teléfonos móviles de los usuarios del servicio. Los usuarios realizarán la conexión a la plataforma preferentemente por medio de un punto de acceso Wi-Fi, aunque también se contempla su acceso por medio de GPRS o UMTS. Si el acceso se realiza por medio de una de las alternativas con coste por volumen de tráfico, como GPRS o UMTS, será interesante conocer cuál puede ser este volumen de tráfico que se genera al jugar a uno de los videojuegos de la plataforma, y por tanto poder conocer qué coste asociado conllevaría para el usuario el uso de la plataforma desde su terminal.

Sin embargo, el coste que conlleva el uso de una de las alternativas de conexión de pago para disfrutar de la plataforma no es el único motivo que existe para realizar estas mediciones de tráfico. Incluso en las alternativas de conexión gratuitas, como Wi-Fi, es interesante conocer la cantidad de tráfico que genera jugar a uno de los videojuegos de la plataforma, para así poder estimar el tiempo de transferencia medio entre la realización de una acción por parte del usuario y la presentación de la nueva localización a éste.

Para realizar las medidas del tráfico generado por la aplicación en el caso de partidas individuales se hizo uso de un ordenador en el que se ejecutó el servidor de la aplicación. También se instalaron los emuladores de la Serie 60 de Nokia y Sun Wireless Toolkit para poder acceder desde ellos a la plataforma. En lugar de realizar la conexión desde los emuladores a la dirección IP local del ordenador, se usó la IP externa de éste, de modo que se pudiese medir el tráfico generado a través del interfaz de la tarjeta de red del ordenador.

Las medidas se realizaron con el analizador de protocolos WireShark, que tiene la particularidad de ser software libre, como el resto de programas utilizados en este proyecto.

Para las medidas realizadas para partidas multijugador se hizo uso de un segundo ordenador situado en una red local con el primero, sobre el que se instalaron también los emuladores para ejecutar la aplicación. De este modo, utilizando el analizador de protocolos fue posible ver el tráfico que se generaba en uno de los clientes cuando el otro compartía localización con él y realizaba un cambio en el modelo. Para ello se usaron los filtros de IP y puerto que proporciona WireShark, para poder ver el tráfico dirigido a uno solo de los dos clientes.

5.1.2 MEDICIONES REALIZADAS

Se pretende que las medidas a realizar sean independientes de las particularidades del usuario del servicio, que puede tener preferencia por explorar todas las acciones en una determinada localización, realizar un juego pausado, tener tendencia a favorecer determinadas acciones sobre otras, poseer conocimientos previos de este tipo de videojuegos, etc. Por ello, se realizaron las medidas atendiendo a las diferentes acciones posibles y a las consecuencias que generan.

Al realizar las medidas con WireShark se capturó todo el tráfico generado en el ordenador en un determinado espacio de tiempo. Este tráfico capturado se visualizaba a continuación con la ayuda de un filtro que permitía extraer y mostrar únicamente el tráfico interesante para las medidas a realizar. Por ello, se usó un filtro que sólo dejaba pasar el tráfico TCP generado en los dos puertos utilizados por la aplicación. En el caso de las medidas para las partidas multijugador, también se realizó un filtrado en función de la dirección IP, para poder centrar la atención únicamente en uno de los clientes. Por último, se visualizó el tráfico con unas gráficas de entrada y salida que proporciona WireShark, como la que se puede ver en la Figura 5.1.

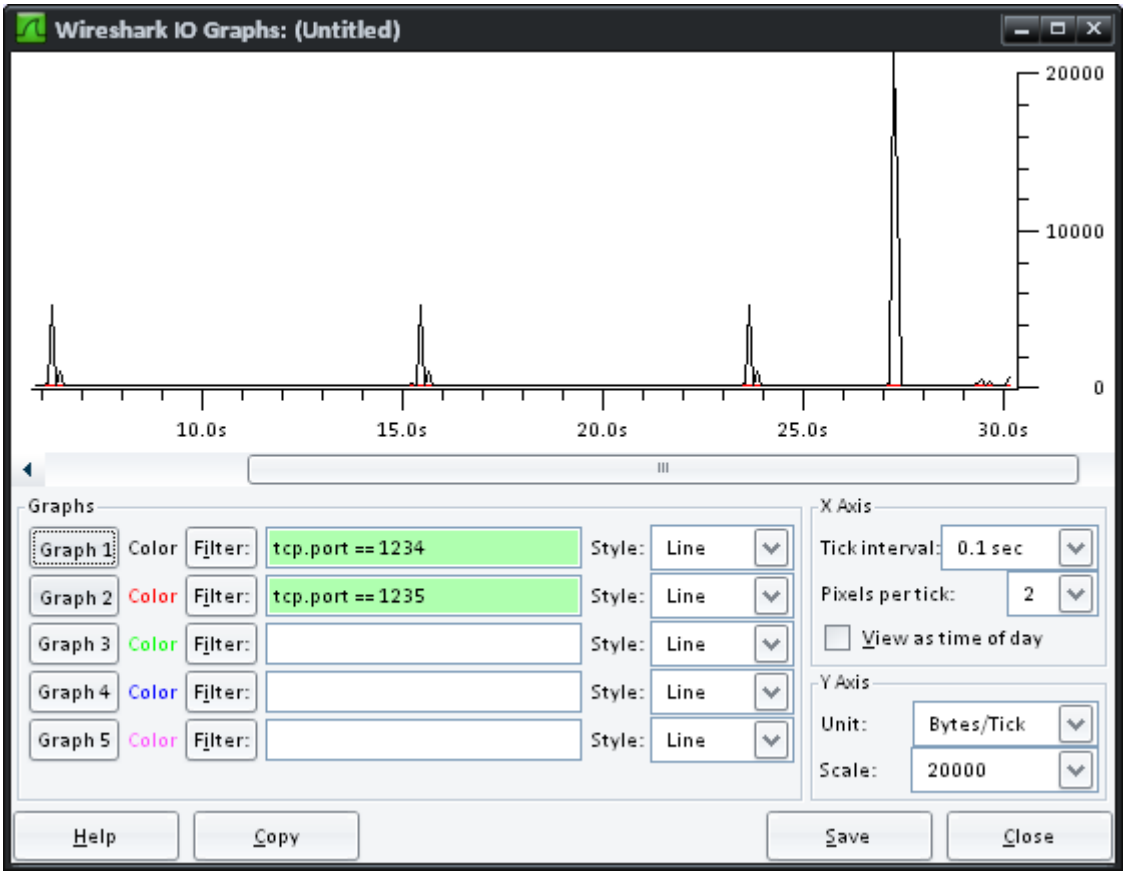


Figura 5.1 Ejemplo de tráfico generado por la aplicación

Para cada tipo de interacción se realizaron varias medidas (entre 5 y 10), de forma que se promediara la información obtenida y se obtuviera una estimación lo suficientemente representativa del tráfico real generado.

En cualquier caso, hay que reseñar que todas las medidas realizadas simplemente pretenden ser orientativas, y que hay que tener en cuenta que ciertos efectos no se pudieron tener en cuenta, como por ejemplo la compresión que se realiza de los datos antes de transferirlos por el medio radio, lo que hace que el tráfico real sea menor que el estimado. También hay que tener en cuenta que todas las mediciones son dependientes por completo del videojuego sobre el que se realicen, y más en concreto del tamaño medio de sus imágenes, y en menor medida, de la historia descrita en los archivos de configuración XML.

Se realizaron las medidas sobre el videojuego “AO”, una adaptación de “La Aventura Original” realizada en [15] sobre la que se llevaron a cabo las modificaciones necesarias para adaptarla a las posibilidades que ofrece la nueva plataforma, comentadas en el capítulo 3 de este documento. Las imágenes modificadas de este videojuego tienen un

tamaño medio de 17,9 KBytes. Este peso es muy importante a la hora de calcular el tráfico que se generará al jugar, ya que la mayor parte de ese tráfico será producido por la transferencia de imágenes. Sin embargo, hay que garantizar una cierta calidad gráfica que mantenga satisfechos a los usuarios.

En la Figura 5.2 se muestran las diferentes medidas realizadas y el valor promedio obtenido del tráfico generado por cada proceso. Hay que reseñar que en el tráfico promedio se incluye tanto tráfico de subida como de bajada, ya que la tarificación en las alternativas de conexión de pago a la plataforma (GPRS, UMTS) se realiza conforme al volumen de tráfico generado, independientemente de que éste sea de subida o bajada.

Tipo de proceso	**Tráfico promedio (KBytes)**
Acceso a la lista de videojuegos	0,8
Salir de la aplicación	0,6
Acceso a la presentación del videojuego	45,8
Acceso a primera localización tras crear partida	47,9
Acceso a primera localización tras entrar en partida (continuar)	41,9
Acceso a primera localización tras entrar en partida (reiniciar)	48,1
Ir a… nueva localización	36,3
Ir a… misma localización	5,1
Acción cuyas consecuencias llevan a nueva localización	36,3
Acción cuyas consecuencias mantienen la localización actual	5,2
Morir (acción que produce la muerte)	27,6
Finalizar videojuego con éxito	46,3
Petición de actualización por cambios en el contexto de juego	0,5
Acción de otro usuario en partida multijugador que provoca actualización cambiando de localización	36,9
Acción de otro usuario en partida multijugador que provoca actualización sin cambiar la localización	5,7

Figura 5.2 Medidas de tráfico realizadas

Como se puede ver en la tabla, hay una gran diferencia entre el tráfico que genera una acción si provoca un cambio en la imagen que ilustra a la localización o si no lo hace. Esto

es debido a que la última imagen utilizada en el *MIDlet* se guarda en un *buffer*, y si al consultar al servidor cuál va a ser la nueva imagen tras realizar una acción ocurre que va a ser la misma, se aprovecha la imagen de dicho *buffer* y no se vuelve a realizar la transferencia. Esta optimización permite reducir el tráfico de 36,3 KB a 5,2 KB por cada vez que se realice una de estas acciones. Además, este acceso a la imagen ya cargada es algo que ocurre muy frecuentemente, ya que la inmensa mayoría de acciones generan consecuencias que no van a cambiar la imagen que ilustra la localización actual, con la excepción de "Ir a" y las acciones que tengan como consecuencia directa modificar la localización completa (con "<reacCambiaAlt>") o provocar la muerte del personaje. El resto de la información referente al entorno de juego, entre la que están fragmentos de archivos XML, objetos, personajes y alertas de texto; se transfieren cada vez que se realiza una acción, generando ese tráfico promedio de 5,2 KB.

En cuanto a lo parecido del tráfico generado independientemente de la acción realizada (siempre que no se compare el tráfico que genera una acción que provoca un cambio en la imagen de la localización con el que genera una acción que mantiene la misma imagen) es debido a que la mayor parte del tráfico se produce por la transferencia de las imágenes y los fragmentos de archivos XML correspondientes a la localización en la que el usuario se encuentra, por lo que el aporte de los distintos mensajes y alertas que provocan una u otra acción al tráfico total es prácticamente imperceptible.

Se han realizado medidas separadas para las acciones que conllevan la muerte del jugador debido a que la localización que muestra la muerte es diferente al resto de localizaciones del videojuego, lo cual se aprecia simplemente viendo el tráfico que se genera.

En el ámbito multijugador, las acciones realizadas por otro jugador que provocan una recarga en el contexto de juego generan un tráfico que es similar al que se produce al seleccionar una acción en el caso de partida individual, más el que corresponde a los mensajes de petición de actualización que intercambia el protocolo desarrollado con el cliente, que al ser mensajes de texto sólo contribuyen en 0,5 KB de promedio al tráfico generado. Además, este tráfico será únicamente de bajada, ya que el envío de los mensajes de texto y la información se realiza desde el servidor al cliente exclusivamente.

5.2 PLAN DE PRUEBAS DETALLADO

En este apartado se detallan las pruebas realizadas a la aplicación para verificar su correcto funcionamiento ante las distintas situaciones que se pueden plantear. No obstante, las pruebas aquí reflejadas son tan solo un resumen representativo de la multitud de pruebas que se han tenido que realizar durante la fase de diseño de la aplicación.

5.2.1 PRUEBAS CON EMULADORES

En primer lugar se realizaron una serie de pruebas con la idea de verificar el correcto funcionamiento general de la aplicación. Para ello se hizo uso de los dos emuladores utilizados en el proyecto: S60 de Nokia y Wireless Toolkit de Sun. Debido a que el funcionamiento del emulador S60 de Nokia se aproxima más al comportamiento de un móvil real, éste fue el emulador preferido para realizar las pruebas de este apartado en las que se estaba en un contexto de partida individual. El servidor de la aplicación se ejecutó en el mismo ordenador en el que se tenían los emuladores, a modo de una primera aproximación para ver el comportamiento de la aplicación.

5.2.1.1 Pruebas en partida individual

Las pruebas realizadas en un contexto de partida individual se mencionan a continuación:

- Acceso a la aplicación en el emulador.
- Petición de introducir IP y puerto del servidor si se intenta acceder al servicio por primera vez sin haberlos configurado previamente.
- Guardado de los datos de IP y puerto del servidor si se ha accedido anteriormente al servicio.
- Presentación de alerta informativa al usuario si introduce una IP o puerto no válidos, así como la posibilidad de volver a introducir estos datos en ese caso.
- Acceso a la lista de videojuegos de la plataforma.
- Correcto abandono de la partida desde la lista de videojuegos.
- Acceso a la presentación de uno de los videojuegos de la plataforma.
- Correcto abandono de la partida desde la presentación del videojuego.
- Correcta creación de partida individual (registro), generando la carpeta con el nombre del usuario y copiando los archivos necesarios.

- Envío del mensaje de error al usuario si intenta registrar una partida individual con un nombre de usuario que ya estaba registrado.
- Notificación de error al usuario si intenta registrar una partida individual en la que no coinciden los dos campos de contraseña en el formulario de registro.
- Notificación de error al usuario si intenta registrar una partida individual dejando alguno de los campos del formulario de registro vacío.
- Notificación de error al usuario si intenta registrar una partida individual utilizando caracteres extraños.
- Acceso al videojuego creando una nueva partida individual.
- Correcta identificación de usuario registrado en partida individual.
- Envío del mensaje de error al usuario si intenta identificarse como usuario registrado cuando no es cierto.
- Envío del mensaje de error al usuario si intenta identificarse como usuario registrado con una contraseña que no corresponde a ese usuario.
- Notificación de error al usuario si intenta identificarse como usuario individual dejando alguno de los campos del formulario de identificación vacío.
- Acceso a partida individual en el videojuego retomando la partida por donde se dejó la última vez que se jugó.
- Acceso a partida individual en el videojuego reiniciando los progresos desde el principio.
- Dos usuarios jugando cada uno de ellos a una partida individual, de forma que no interfieren el uno con el otro.
- Correcta presentación de los diferentes contenidos, imágenes, textos, formularios, etc.
- Funcionamiento de los diferentes formularios presentados, permitiendo al usuario poder visualizar todas las opciones.
- Correctas modificaciones en los formularios para adecuarlos a la situación actualizada del juego.
- Movimiento del usuario por las diferentes localizaciones del mapa de un videojuego usando el comando “Ir a”.
- Presentación de la misma imagen en una localización sin necesidad de traerla de nuevo del servidor si se ha realizado una acción cuyas consecuencias hagan que se cargue esa misma imagen en el cliente.

- Coger y dejar objetos que se muestran al usuario, modificando correctamente el inventario.
- Coger un objeto y dejarlo en otra localización diferente, mostrando de forma correcta el texto descriptivo del mismo en la nueva localización.
- Imposibilidad de coger objetos cuyo atributo “cogible=no”.
- Examinar un objeto, mostrando su texto a modo de alerta al usuario.
- Abrir y cerrar objetos que permitan dicha acción, y comprobar que modifican las direcciones presentes en el formulario “Ir a” (por ejemplo, al abrir una puerta, se activa la posibilidad de pasar por ella).
- Usar objetos unos con otros siempre y cuando esté especificado en los archivos de configuración, de lo contrario, mostrar alerta avisando de que no es posible.
- Imposibilidad de usar objetos cuyo atributo “usable=no”.
- Correcta generación de consecuencias tras seleccionar las diferentes acciones que las provocan.
- Correcta modificación de los archivos de configuración del usuario ante una acción o movimiento a otra localización.
- Hablar con un NPC navegando por las distintas frases que el usuario puede elegir decirle, y ver las respuestas del NPC ante éstas.
- En un diálogo con un NPC, comprobación de que se puede abandonar la conversación en cualquier momento y retomarla por el punto por el que iba más tarde.
- Hablar con un NPC hasta llegar al fin de la conversación, tras el cual se producen una serie de consecuencias que afectan al contexto de juego.
- Imposibilidad de hablar con un objeto cuyo atributo “npc=no”.
- Dar objeto a un NPC, si se ha definido que el NPC va a aceptar ese objeto en los archivos de configuración.
- Imposibilidad de dar un objeto a un NPC si no tiene definido en los archivos de configuración que acepta dicho objeto.
- Correcta modificación del texto e imagen de una localización por medio de una consecuencia de tipo “<reacCambiaAlt>”.
- Entrada en una localización que tiene un temporizador definido, lo que provoca que se active y, tras un cierto tiempo definido en los archivos de configuración, se notifique al usuario de la existencia de un evento temporizado en esa localización.

- Espera en una localización con temporizador definido hasta que se llega al último de los temporizadores, lo que causa una serie de consecuencias en el contexto de juego.
- Puesta a cero del temporizador si se abandona la localización en la que estaba definido.
- Muerte del usuario al realizar una acción que tiene como consecuencia definida la muerte del personaje que la realice.
- Presentación de la localización que indica la muerte del jugador.
- Presentación de la localización de fin de la aventura.
- Correcto reinicio del juego tras la muerte o fin de la aventura, si el usuario lo desea.
- Correcta salida desde el juego a la lista de videojuegos de la plataforma.

5.2.1.2 Pruebas en partida multijugador

Para el caso de las pruebas realizadas en el contexto de partida multijugador hubo que utilizar varias instancias del emulador Sun Wireless Toolkit, debido a la imposibilidad de ejecutar más de una instancia del emulador S60 de Nokia en una misma máquina y a los exigentes requisitos de memoria que éste requiere. A continuación se nombran las pruebas adicionales realizadas en un contexto multijugador:

- Correcta creación de partida multijugador (registro), generando la carpeta con el nombre de la partida y copiando los archivos necesarios.
- Envío del mensaje de error al usuario si intenta registrar una partida multijugador con un nombre de partida que ya estaba registrado.
- Notificación de error al usuario si intenta registrar una partida multijugador en la que no coinciden los dos campos de contraseña en el formulario de registro.
- Notificación de error al usuario si intenta registrar una partida multijugador dejando alguno de los campos del formulario de registro vacío.
- Notificación de error al usuario si intenta registrar una partida multijugador utilizando caracteres extraños.
- Acceso al videojuego creando una nueva partida multijugador.
- Correcta identificación de usuario registrado en partida multijugador y acceso al videojuego por el punto en que lo dejó la última vez que jugó.
- Envío del mensaje de error al usuario si intenta identificarse como usuario registrado en una partida en la que no lo está.

- Envío del mensaje de error al usuario si intenta identificarse como usuario registrado con una contraseña que no corresponde a ese usuario.
- Notificación de error al usuario si intenta identificarse como usuario en partida multijugador dejando alguno de los campos del formulario de identificación vacío.
- Acceso de dos jugadores a una misma partida multijugador.
- Comprobación de que todas las acciones realizadas por un jugador en una localización causan una recarga de dicha localización en el resto de jugadores presentes en ella (por ejemplo, si un jugador entra en una localización con más jugadores, o si coge un objeto en una localización en la hay más jugadores).
- Comprobación de que las acciones realizadas por un jugador que tengan efecto sobre otras localizaciones, causan una recarga del contexto de juego en los usuarios que se encuentren en esas localizaciones afectadas por las consecuencias de dichas acciones.
- Verificación de que, en cualquier momento, los usuarios presentes en una localización del juego están recogidos en los formularios de las acciones “Dar a” y “Hablar con”.
- Hablar con un usuario presente en la misma pantalla, lo que provoca una recarga en el contexto de juego de dicho usuario y que le aparezca el mensaje de texto enviado.
- Dar un objeto a un usuario presente en la misma pantalla, lo que provoca una recarga en el contexto de juego de dicho usuario y que este objeto se incorpore a su inventario.
- Intentar hablar con un usuario presente en la misma pantalla o darle un objeto, y mientras se navega por los menús, este usuario sale de la pantalla, lo que causa una recarga en el contexto de juego en esa pantalla y hace que no se lleve a cabo la acción.
- Realizar con dos jugadores de forma simultánea alguna acción que sólo se pueda realizar una vez tras lo cual cambia el contexto de juego (por ejemplo, coger un objeto, lo que provoca que ese objeto desaparezca y no pueda ser cogido por más jugadores), y comprobar que, efectivamente, sólo el usuario que accede primero al servidor consigue coger el objeto, mientras que al otro se le muestra un mensaje informándole de la imposibilidad de realizar la acción al no encontrarse ya el objeto en el escenario.

- Intentar hablar con un usuario presente en la misma pantalla o darle un objeto, y simultáneamente al envío de la acción, el usuario receptor abandona la pantalla, lo que hace que se le muestre un mensaje al otro usuario informándole de que no se ha podido realizar la acción porque el usuario receptor ha abandonado la localización.
- Entrada de dos jugadores en una localización con un temporizador definido, para comprobar que los temporizadores son locales a los usuarios y que se notifica de los cambios a cada uno cuando expira su temporizador particular.
- Correcto reinicio del juego llevando al usuario a la primera localización tras la muerte o fin de la aventura, si el usuario lo desea.
- Acceso de más de dos jugadores a una misma partida multijugador.
- Seleccionar la opción "Hablar con" en una localización en la que haya más de dos jugadores de forma simultánea, lo que provoca que aparezca una nueva opción de "hablar a todos", y comprobar que, efectivamente, el mensaje le llega a todos los usuarios de la localización.

5.2.2 PRUEBAS CON DISPOSITIVOS MÓVILES REALES

Finalmente y tras haber pasado satisfactoriamente las pruebas del emulador, se realizaron una serie de pruebas en móviles reales. Para ello, se buscaron terminales móviles que dispusieran de conectividad Wi-Fi, y se montó un punto de acceso inalámbrico para que pudieran conectarse a través de él al servicio de videojuegos conversacionales.

Para estas pruebas se usaron un par de terminales móviles Nokia 5800, pertenecientes a la Serie 60 de Nokia, 5ª edición. Asimismo, también se usó un móvil Nokia E65, perteneciente a la Serie 60 de Nokia, 3ª edición. Todos estos terminales disponen de conectividad Wi-Fi.

En estos terminales se realizaron las pruebas descritas en el apartado 5.2.1, prestando una especial atención a los efectos de la realización de acciones de forma simultánea en un contexto de partida multijugador, para lo que se utilizaron los dos terminales Nokia 5800 en un intento de hacer que las condiciones fueran lo más parecidas posibles para realizar pruebas en el caso peor de un acceso simultáneo desde ambos móviles, por ejemplo intentando coger un objeto desde ambos terminales en el mismo instante.

Se planteó la realización de pruebas de carga del servidor para ver su comportamiento en la situación de múltiples usuarios jugando de forma simultánea, pero finalmente no se llevaron a cabo debido a la gran cantidad de terminales móviles que serían necesarios para obtener datos relevantes, así como a la total dependencia que existiría de la máquina física sobre la que se ejecutara el servidor y de sus características como procesador, memoria, etc.

CAPÍTULO 6: CONCLUSIONES Y LÍNEAS FUTURAS

6.1 CONCLUSIONES FINALES

El objetivo de este proyecto fin de carrera ha sido desarrollar un motor de videojuegos conversacionales para dispositivos móviles que proporcionara la posibilidad de jugar simultáneamente con otros jugadores en la misma partida. Este motor presenta al usuario una localización de la historia ilustrada por una imagen, a la que acompaña un texto descriptivo y una lista acciones posibles a realizar. Estas acciones se realizan sobre los objetos o personajes disponibles en el escenario, de modo que una vez se haya seleccionado la acción, se envía al servidor, que es el encargado de procesarla, llevarla a cabo si es posible, y presentar al jugador el nuevo contexto de juego en el que se encuentra tras considerar las consecuencias de su acción.

El motor propuesto se basa en el desarrollado en [15] siguiendo la arquitectura MVC. Para la realización del nuevo motor hubo que adquirir las competencias necesarias en las tecnologías usadas en el motor original (JSP, *servlets*, etc.), con el fin de lograr una profunda compresión de la aplicación antes de poder modificarla y adaptarla a los requisitos pedidos.

Una vez conseguido esto, fue necesario eliminar toda la parte correspondiente a la vista del motor original y desarrollar un *MIDlet* J2ME que presentara el interfaz de juego al usuario e interaccionara con él. Este *MIDlet* tenía que tener la característica de que todos sus elementos debían ser configurables en base a unos archivos XML que residen en el servidor, para así poder dar cabida al sencillo desarrollo de nuevos videojuegos para la plataforma, para el que no es necesario tener conocimientos de programación; o incluso a realizar adaptaciones a otros idiomas sin más que modificar los archivos de configuración XML. Es por ello que se incorporó la lógica necesaria en el cliente para poder realizar la extracción y tratamiento de los datos del juego contenidos en estos archivos XML por medio del uso de un ligero *parser* XML.

De forma paralela, se adaptó el servidor formado por *servlets* Java que implementaban su inteligencia y archivos JSP que generaban de forma dinámica los archivos HTML que hacían de interfaz con el usuario, hasta convertirlo en una única aplicación Java que realizase la misma funcionalidad del motor original.

Asimismo, eliminado el mecanismo de comunicación del motor original por medio del navegador web incluido en el terminal móvil, fue necesario diseñar e implementar un protocolo propio que permitiese la comunicación entre el servidor y los clientes. Este protocolo se basa en un envío de mensajes entre cada cliente y el servidor por medio de sockets, seguido del intercambio de la información que corresponda. En dichos mensajes se indica el estado en el que se encuentra el sistema y, por tanto, la información que han de intercambiar los extremos de la comunicación. Así, se generó una máquina de estados que sustituye a la comunicación por medio de WAP/HTML establecida en el motor original.

En cuanto a la forma de conexión del móvil al servidor, se propone el uso de un móvil con conectividad Wi-Fi que pueda conectarse a uno de los múltiples puntos de acceso inalámbrico que proporcionan un acceso libre y gratuito a Internet en cualquier ciudad en la actualidad. En el caso de que esto no sea posible, se puede llevar a cabo la conexión por medio de GPRS o UMTS, que son tecnologías con una amplia expansión en todo el territorio y que cada día tienen un coste más razonable.

Con la creación del cliente J2ME, la adaptación del servidor del motor original formado por *servlets* y archivos JSP a un servidor Java, y el desarrollo de un protocolo que permitiera la comunicación entre ambos extremos, se consiguió tener una versión preliminar de la aplicación que llevase a cabo la funcionalidad del motor original. Hecho esto, se tenía una plataforma capaz de soportar a varios usuarios jugando cada uno a una instancia del videojuego, de forma independiente y sin comunicación entre ellos. Por lo que lo siguiente fue abordar el diseño que incorporase la funcionalidad multijugador en la plataforma, para poder tener a varios jugadores simultáneos jugando a una misma partida del videojuego, de forma que las acciones de uno de ellos tuvieran un efecto inmediato sobre el resto de jugadores de la partida.

Por tanto, hubo que rediseñar por completo la parte de control, para lo que fue necesario hacer que todos los usuarios que forman parte de una misma partida compartieran el acceso a un conjunto de archivos de configuración XML. Así, fue necesario afrontar los

problemas propios que derivan del desarrollo de aplicaciones concurrentes. Se implementó un mecanismo de regulación de paso que solo permitiera el acceso autorizado y ordenado de los usuarios a estos archivos. Asimismo, también se implementó un sistema que protegiese la coherencia de los archivos XML que forman el modelo, para lo cual se utilizó el patrón observador. De esta forma, cuando un usuario en una partida multijugador realiza una acción que cambia de alguna forma el contexto de juego en el que se encuentra (ya sea cambiando los objetos, el propio escenario, o a los jugadores que en él se encuentran) se notifica al resto de jugadores que están en las localizaciones afectadas por los cambios para que actualicen su información del entorno en el que se encuentran. De este modo se garantiza que los usuarios siempre tengan una versión actualizada del contexto de juego reflejado en los archivos de configuración XML. Para esto, fue necesario modificar el protocolo desarrollado e incorporar un nuevo canal por donde se llevarían a cabo las notificaciones de actualización desde el servidor a los clientes por medio de una serie de mensajes, seguidos de los datos pertinentes.

Otro problema que se planteó durante el desarrollo de la versión multijugador de la aplicación fue qué ocurriría en el evento de que se produjesen dos acciones de forma simultánea por parte de dos jugadores sobre un mismo objeto, antes de que hubiera dado tiempo al sistema a actualizar al resto de usuarios de la partida con la nueva situación. Se tuvo que prever esa contingencia protegiendo la realización de acciones en el servidor, de modo que se comprobase de nuevo si era posible realizar la acción elegida en el móvil antes de intentar llevarla a cabo, y si no era el caso, que se notificase al usuario de esta eventualidad.

Por otra parte, se incluyeron nuevas acciones y funcionalidades en el motor original, entre ellas:

- La presencia de NPC con una cierta inteligencia, de modo que se pueda conversar con ellos por medio de diálogos prediseñados incorporados en los archivos XML de configuración. Para ello se usa la nueva acción “Hablar con”. Por medio de la conversación con NPC se pueden conseguir pistas para seguir avanzando en la historia, o incluso objetos cruciales para seguir progresando. Así, estos NPC aportan una nueva dimensión a los videojuegos desarrollados para la plataforma, proporcionando una mayor interactividad y una gran cantidad de posibilidades para realizar un buen guión que haga que la aventura enganche al jugador.

- Al incorporarse los NPC, también se vio apropiado proporcionar una nueva acción "Dar objeto a", que permitiese a los jugadores darle objetos a los NPC y así lograr que se produjesen una serie de consecuencias sobre el escenario y los propios jugadores.
- Algunas de las nuevas acciones definidas también cobran un nuevo sentido en el caso de que se esté en una partida multijugador. Es el caso de la acción "Hablar con", que si se aplica a otro jugador en una partida colectiva, permite al usuario enviar mensajes de texto para comunicarse con otros jugadores que se encuentren en su misma localización. En el caso de que haya más de dos jugadores en la misma localización, se podrá hablar a todos.
- Como en el caso anterior, la acción "Dar objeto a", en el caso de una partida multijugador, se puede aplicar a otros jugadores para entregarles objetos que se tengan en el inventario, y así poder colaborar entre varios jugadores para lograr un fin común y seguir avanzando en la historia.
- La creación de un nuevo tipo de localización, que tiene como particularidad tener definido un tiempo límite, transcurrido el cual ocurren una serie de consecuencias. Así, el servidor controla por medio de un temporizador el tiempo transcurrido en esas localizaciones, y cuando éste expira, notifica a los usuarios de esa eventualidad y, en su caso, genera las consecuencias apropiadas. Por ejemplo, si un usuario está de pie cerca de un volcán, se puede establecer un tiempo tras el que se le avisa de que hace demasiado calor y debería abandonar la localización. Si el usuario ignora las advertencias, se puede causar la muerte del personaje al no poder soportar la elevada temperatura.
- La modificación de una localización completa por medio de una nueva consecuencia del tipo "<reacCambiaAlt>", que sustituye la localización actual por otra completamente diferente y hace que este cambio se propague al resto de usuarios si se está en una partida multijugador, como por ejemplo encender la luz en una habitación en la que estaba oscuro, tras lo cual hay que mostrar la nueva localización en la que ya se puede ver.

En cuanto al modelo, fue necesario incluir en los archivos de configuración XML en los que está descrito el videojuego una serie de etiquetas y atributos que recogieran las nuevas posibilidades ofrecidas por el motor.

Como en todo proceso de ingeniería, conforme se iba avanzando en el desarrollo fueron surgiendo inconvenientes asociados a las decisiones tomadas, que en ciertos casos hubo que revisar y, en otros, hicieron que se tuviera que reconsiderar el camino elegido y hubiera que elegir una alternativa mejor a la vista de las dificultades surgidas. Sin embargo, es la habilidad principal del ingeniero tener la capacidad para adaptarse a los cambios y explorar nuevas alternativas, algunas de las cuales vienen dadas por personas que contribuyen aportando una perspectiva externa al proceso, sin la cual la superación de esos obstáculos habría sido mucho más complicada.

6.2 LÍNEAS FUTURAS

No obstante, a pesar de haber cumplido todos los requisitos planteados inicialmente, a lo largo del desarrollo del proyecto han quedado abiertas varias posibles líneas de expansión o incorporación de mejoras en el sistema:

- Desarrollar un entorno gráfico para la confección de nuevas aventuras conversacionales, evitando de esa forma que el usuario tenga que manejar directamente los archivos XML que conforman cada videojuego. Así, el usuario administrador tan solo tendría que ejecutar la aplicación de creación de nuevas aventuras conversacionales y servirse de una librería de símbolos que se correspondieran con los elementos que conforman el mapa en este tipo de juegos (localizaciones, posibles direcciones a tomar en una localización, objetos, etc.) para que la aplicación generase de forma automática los archivos XML correspondientes al videojuego descrito.
- Al tener un mundo persistente, en el que los personajes controlados por los usuarios aparecen en la localización en la que están, incluso si no hay un jugador que los esté controlando en ese momento, se podría implementar un modo *offline*, según el cual los personajes de los jugadores que no se encuentren en el juego en ese momento, en lugar de no responder, puedan tener una serie de diálogos y acciones definidas, para dotar de más realismo al entorno. Para ello se puede aprovechar el diseño que se realizó en este proyecto de los NPC inteligentes, e incorporarlos en lugar del control humano cuando el jugador se desconecte.
- Incorporación del comercio en los videojuegos de la plataforma por medio de la creación de una moneda virtual, que permita a los usuarios comprar, entre otros, objetos necesarios para avanzar en la aventura en determinados establecimientos. Para ello, el jugador podría ir ganando dinero virtual o créditos por medio de las

acciones realizadas en el videojuego, que más tarde podría ser intercambiado por objetos o utilizado para comprar la ayuda de algún NPC.

BIBLIOGRAFÍA Y REFERENCIAS

BIBLIOGRAFÍA

Agustín Froufe Quintas y Patricia Jorge Cárdenes, "J2ME Java 2 Micro Edition. Manual de usuario y tutorial", editorial Ra-Ma, edición 2004. Manual de consulta para J2ME.

Manuel J. Prieto, "Desarrollo de juegos con J2ME. Java 2 Micro Edition", editorial Ra-Ma, edición 2005. Manual de consulta para J2ME enfocado al desarrollo de juegos.

Israel Pastrana Vicente, Carlos Araiz García y Víctor Martínez Conte, "Java 2 v5.0. Manual Imprescindible", editorial Anaya, edición 2005. Manual de referencia de Java.

Felipe Lima Díaz, "Java 2 v5.0. Manual Avanzado", editorial Anaya, edición 2006.

Grady Booch, James Rumbaugh y Ivar Jacobson, "El Lenguaje Unificado de Modelado (UML)", editorial Pearson Addison Wesley, edición 2006.

Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes y Doug Lea, "Java Concurrency in Practice", editorial Addison Wesley Professional, edición 2006.

http://cinderalla.iespana.es/invierno/introj2me.pdf. Manual de introducción a J2ME.

Artículo del portal Wikipedia sobre los patrones de diseño software:
http://es.wikipedia.org/wiki/Patrón_de_diseño

Artículo del portal Wikipedia sobre el patrón Observador:
http://es.wikipedia.org/wiki/Observer_(patrón_de_diseño)

Artículo del portal Wikipedia sobre XML:
http://es.wikipedia.org/wiki/Extensible_Markup_Language

Artículo del portal Wikipedia sobre XSD:
http://es.wikipedia.org/wiki/XML_Schema

Análisis y comparación de distintos *parsers* XML en la web de Sun:
http://developers.sun.com/mobility/midp/articles/parsingxml/

Tutorial con ejemplos de distintos tipos de diagramas UML:
http://docs.kde.org/kde3/es/kdesdk/umbrello/uml-elements.html

Artículo sobre redes 2.5G y GPRS:
http://voicendata.ciol.com/content/columns/fromcell/101052501.asp

Artículo del portal Wikipedia sobre UMTS:
http://es.wikipedia.org/wiki/Universal_Mobile_Telecommunications_System

Artículo sobre las posibilidades que ofrece Ant para construcción y compilación de software:
http://www.javamexico.org/blogs/benek/introduccion_apache_ant

REFERENCIAS

[1] Killer List of Videogames, web dedicada a mantener una amplia base de datos de máquinas recreativas.
http://www.klov.com/game_detail.php?game_id=9074

[2] Artículo relacionado con la crisis del sector de los videojuegos de 1983 de la versión inglesa del portal Web de la Wikipedia.
http://en.wikipedia.org/wiki/North_American_video_game_crash_of_1983

[3] Web de la aDeSe (Asociación española de Distribuidores y Editores de Software de Entretenimiento).
http://www.adese.es/web/main.asp

[4] Noticia aparecida el 30 de Septiembre de 2009 en el portal multimedia de negocios y nuevas tecnologías Baquia, titulada "Los videojuegos ya suponen más del 50% del ocio interactivo en España".
http://www.baquia.com/noticias.php?id=15330

[5] Noticia publicada el 30 de Septiembre de 2009 en el sitio web especializado en tecnología tuexperto.com, titulada "El videojuego se confirma como el tipo de ocio más potente, rentable y lucrativo".
http://www.tuexperto.com/2009/09/30/el-videojuego-se-confirma-como-el-tipo-de-ocio-mas-potente-rentable-y-lucrativo/

[6] Noticia del 12 de Febrero de 2009 aparecida en el portal de noticias CNET News, titulada "Aumenta el uso de WiFi en los teléfonos móviles" (*Mobile phone Wi-Fi usage on the rise*).
http://news.cnet.com/8301-1035_3-10162711-94.html

[7] Noticia aparecida el 24 de Julio de 2008 en el magazine de noticias empresariales, marketing y negocios en Internet InfonoNews, titulada "2 años y 5 meses es el tiempo promedio que los españoles utilizan el móvil, luego lo cambian".
http://www.infonos.com/2-anos-y-5-meses-es-el-tiempo-promedio-que-los-espanoles-utilizan-el-movil-_i9833.html

[8] Noticia del periódico El País del 8 de Mayo de 2008 con el título "La venta de videojuegos en el móvil aumenta".
http://www.elpais.com/articulo/internet/venta/videojuegos/movil/aumenta/elpeputec/20080508elpepunet_5/Tes

[9] Club de Aventuras AD. Página de la comunidad hispana de Aventuras Conversacionales y Relatos Interactivos.
http://www.caad.es

[10] Web asociada al Club de Aventuras AD en la que se recogen librerías, herramientas y documentación para la elaboración de aventuras conversacionales en español. También se recogen juegos desarrollados usando

estas herramientas por aficionados al género, e intérpretes para poder jugarlos en casi cualquier plataforma.
http://www.caad.es/informate/

[11] Web en la que se encuentra alojado el fanzine de la SPAC (Sociedad para la Preservación de las Aventuras Conversacionales), dedicado a proporcionar noticias y artículos sobre las aventuras conversacionales y todo tipo de relatos interactivos.
http://spac.caad.es/

[12] Web de la SPAG (*Society for the Promotion of Adventure Games*), en la que se realizan análisis, entrevistas y artículos sobre ficción textual interactiva y otros tipos de narrativa interactiva, en una publicación online de forma trimestral.
http://www.sparkynet.com/spag/

[13] Entrevista a Rand Miller, creador del juego Myst, publicada el 22 de Mayo de 2001 en el periódico The Spokeman-Review, en la que se referencian las 6 millones de copias vendidas de dicho juego, que en ese momento le convertían en el juego para PC más vendido de la historia.
http://www.spokesmanreview.com/news-story.asp?date=052201&ID=s966647

[14] Gráfica detallando el porcentaje estimado de mercado ocupado por World of Warcraft a fecha de Abril de 2008, extraída de MMOGChart.com, sitio web dedicado a investigar el crecimiento de las suscripciones a juegos multijugador masivos online (del inglés MMOG, *Massively Multiplayer Online Games*).
http://www.mmogchart.com/Chart7.html

[15] Javier González Monroy, "Desarrollo de un motor de videojuegos conversacionales sobre WAP para dispositivos móviles", Proyecto Fin de Carrera, Universidad de Málaga, 2007.

[16] Artículo sobre el origen de los patrones de software y de su precursor, el arquitecto Christopher Alexander.
http://www.usolab.com/wl/2007/06/experiencias-paralelas-christo-1.php

[17] Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides, "Design Patterns: Elements of Reusable Object-Oriented Software", editorial Addison-Wesley, edición 1994. Primer libro que incluyó una serie de patrones para dar solución a problemas comunes en el campo de la ingeniería del software.

[18] Web del *parser* XML para Java Xparse-J.
http://www.webreference.com/xml/tools/xparse-j.html

[19] Web de la distribución de Linux WifiSlax, orientada a la auditoría de seguridad de redes inalámbricas.
http://www.wifislax.com/

[20] Artículo aparecido en el periódico El País el 30 de Julio de 2006 con el título "Wi-Fi por la cara".
http://www.elpais.com/articulo/madrid/Wi-Fi/cara/elpepuespmad/20060730elpmad_13/Tes

[21] Comparativa sobre las distintas tarifas que ofrecen las operadoras móviles para conectarse a Internet desde el móvil.
http://www.xatakamovil.com/movistar/comparativa-tarifas-internet-para-navegar-desde-el-movil

[22] Comparativa sobre las coberturas 3G y 3.5G estimadas por las propias operadoras móviles.
http://bandaancha.eu/tema/1554154/cobertura-gsm3g-35g-movistarvodafoneorangeyoigo

[23] Noticia aparecida en el portal Todomovil.com el 20 de Abril de 2009, en la que se dice que Telefónica ya ha superado el millón de tarifas planas de datos asociadas a dispositivos móviles 3G.
http://www.todomovil.com/servicios-moviles/internet-movil/telefonica-alcanza-el-millon-de-tarifas-planas-de-datos-para-dispositivos-internet-movil.html

[24] Web del IDE NetBeans.
http://www.netbeans.org/

[25] Web de la herramienta automatizada de construcción y compilación Ant.
http://ant.apache.org/

[26] Web del ofuscador y optimizador para código Java y J2ME ProGuard.
http://proguard.sourceforge.net/

[27] Web del emulador de dispositivos MIDP Sun Wireless Toolkit.
http://java.sun.com/products/sjwtoolkit/download.html

[28] Web de donde se pueden descargar distintas versiones del emulador de los móviles de la Serie 60 de Nokia.
http://www.forum.nokia.com/info/sw.nokia.com/id/6e772b17-604b-4081-999c-31f1f0dc2dbb/S60_Platform_SDKs_for_Symbian_OS_for_Java.html

[29] Web del analizador de protocolos WireShark.
http://www.wireshark.org/

[30] Web de la que se puede descargar el entorno *Java Runtime Environment*, necesario para ejecutar aplicaciones Java.
http://java.sun.com/javase/downloads/index.jsp#jre

[31] Web de Nokia en la que se puede descargar el software Nokia PC Suite.
http://www.nokia.es/soporte/software/pcsuite