Universidad Rey Juan Carlos
Escuela de Ciencias Experimentales y Tecnología
Departamento de Informática, Estadística y Telemática

Tesis Doctoral

Oni: Arquitectura para la relocalización transparente de interfaces de usuario distribuidos y heterogéneos en sistemas ubicuos

Gorka Guardiola Múzquiz
paurea@lsub.org

Director de Tesis:
Francisco J. Ballesteros Cámara
nemo@lsub.org

28 de junio de 2007

Yo, Francisco J. Ballesteros Cámara, Director de la Tesis Doctoral titulada *Oni: Arquitectura para la relocalización transparente de interfaces de usuario distribuidos y heterogéneos en sistemas ubicuos* realizada por Gorka Guardiola Múzquiz, autorizo su defensa.

Francisco J. Ballesteros Cámara
Director de la Tesis Doctoral

Madrid, de de 2006

TESIS DOCTORAL:	Oni: Arquitectura para la relocalización transparente de interfaces de usuario distribuidos y heterogéneos en sistemas ubicuos
AUTOR:	Gorka Guardiola Múzquiz
DIRECTOR:	Francisco J. Ballesteros Cámara

El tribunal nombrado para juzgar la Tesis arriba indicada, compuesto por los siguientes doctores:

PRESIDENTE:	Dr.
VOCALES:	Dr.
	Dr.
	Dr.
SECRETARIO:	Dr.

acuerda otorgarle la calificación de

Madrid, de de 2006

El Secretario del Tribunal

Resumen

El fin de la *computación ubicua* es integrar computadores en objetos comunes y en la infraestructura de forma que desaparezcan de la atención del usuario y se conviertan en herramientas de uso cotidiano. La computación ubicua estudia la posibilidad de un futuro de entornos cotidianos con miles de computadores integrados de forma invisible que reciben el nombre de *entornos ubicuos* [153] [152].

Los entornos ubicuos tienen una serie de características peculiares que dificultan la construcción de sistemas y más particularmente, de interfaces de usuario. Por un lado, son más complejos por la heterogeneidad del hardware y software y la diversidad de dispositivos implicados. Por otro, tienen mayores exigencias en cuanto a la necesidad de adaptación, transparencia de red y automatización.

En la presente tesis se describe una arquitectura novedosa para la construcción de interfaces de usuario adaptadas a entornos ubicuos que resuelve estos problemas.

Esta arquitectura se basa en tres características fundamentales que hacen que las interfaces de usuario puedan adaptarse a estas exigencias y entornos:

- Como interfaz de comunicación entre la interfaz de usuario y los programas cliente se utiliza un **sistema de ficheros** sintéticos en red que los separa y permite que se comuniquen a través de la red en caso necesario. Esto reifica el interfaz y aumenta su programabilidad, lo que permite utilizar, por ejemplo, scripts de shell para controlar la interfaz de usuario. Además, podemos utilizar los mecanismos de protección, de replicación y de transparencia de red de los sistemas de ficheros en red, lo que nos provee

de modelos familiares y bien conocidos para resolver estos problemas, que son complejos y han resultado de difícil solución en el pasado.

- Los objetos presentados por el servidor de interfaces de usuario a los clientes son **widgets, elementos de alto nivel autocontenidos**. Estos elementos son independientes y se relacionan entre sí mediante otros elementos representando relaciones de agrupación y relaciones espaciales. Gracias al alto nivel de abstracción de los elementos implicados y a la separación que ofrece la interfaz de ficheros, se puede adaptar la interfaz de forma transparente a la aplicación para atender las necesidades del usuario. En los entornos ubicuos el usuario se encuentra rodeado de computadores que compiten por su atención. Gracias a la adaptabilidad de nuestra interfaz es más sencillo administrar esta atención, un bien que puede resultar escaso en estos entornos.

- Los mensajes que se intercambian a través del sistema de ficheros, referidos a eventos, y a la manipulación del contenido y relaciones entre widgets son del **mayor nivel de abstracción posible**. Esto permite, además de una mayor flexibilidad de adaptación, la utilización de un menor ancho de banda y recursos de comunicación. En los entornos ubicuos en los que las redes son heterogéneas y las capacidades de comunicación pueden ser reducidas debido a la ausencia de infraestructura, esta propiedad es particularmente útil.

Gracias a estas características, la arquitectura permite la construcción de interfaces de usuario en sistemas heterogéneos y la adaptación, automatización y mantenimiento transparente de vistas en entornos ubicuos de una forma segura.

Las contribuciones de esta tesis son principalmente tres:

- **Identificación de la problemática** asociada a interfaces de usuario en entornos ubicuas.

- **Diseño de una arquitectura** específicamente diseñada para estas interfaces.

- **Validación mediante la comprobación de los requisitos y la implementación** de un prototipo que verifica la viabilidad de la aproximación tomada.

Índice general

Índice general **9**

Índice de figuras **15**

1. Introducción **17**

1.1. Computación ubicua 17

1.2. Plan B 20

1.3. Interfaces de usuario 22

1.3.1. Interfaces de usuario ubicuos 25

2. Descripción del problema y objetivos **27**

2.1. Interoperabilidad entre dispositivos heterogéneos y adaptación a diferentes capacidades 29

2.2. Reflexión y programabilidad 31

2.3. Adaptación modal. Separación de lógica y presentación 33

2.4. Protección 34

2.5. Mantenimiento transparente de vistas y migración 35

2.6. Conclusión 37

2.7. Objetivos 39

3. Estado del arte **41**

3.1. Introducción y terminología 42
3.1.1. Mecanismos generales para la distribución 43
3.1.2. Medio, modo y modalidad 44
3.1.3. Widgets 45
3.1.4. Interfaces de usuario de linea de comando 45
3.1.5. Sistemas de ventanas 46
3.1.6. Manejador de ventanas 48
3.1.7. Interfaces de usuario telescópicos 49
3.1.8. Toolkits 50
3.1.9. Frameworks o infraestructuras de desarrollo 51
3.1.10. Patrones de diseño 53
3.1.11. Programación basada en componentes 53
3.2. Taxonomía de las infraestructuras
para construir interfaces de usuario 54
3.3. Interoperabilidad entre dispositivos
heterogéneos y adaptación a diferentes
capacidades 58
3.3.1. Motif 58
3.3.2. wxWidgets 59
3.3.3. Tcl/Tk 61
3.3.4. GTK 62
3.3.5. Qt 63
3.3.6. AWT 64
3.3.7. Swing 64
3.4. Reflexión y programabilidad 65
3.4.1. OLE, ActiveX, COM, DCOM y CORBA 65
3.4.2. KPARTS 66
3.5. Adaptación modal. Separación entre lógica y
presentación 67
3.5.1. Blit 67

3.5.2. MVC 68
3.5.3. MVP 71
3.5.4. Morphic 72
3.5.5. Tweak 74
3.5.6. Sistemas basados en XML 75
3.5.7. Sistemas basados en modelos 76
3.5.8. Protium 77
3.5.9. Ajax 79
3.5.10. Scgui 80
3.5.11. XML11 80
3.6. Protección 82
3.6.1. Mux 82
3.6.2. $8\frac{1}{2}$ 83
3.6.3. Rio 83
3.6.4. Sam 84
3.6.5. Acme 85
3.7. Mantenimiento transparente de vistas y migración 85
3.7.1. DPS o Display PostScript 86
3.7.2. NeXTStep 87
3.7.3. Quartz 87
3.7.4. NeWS 88
3.7.5. X Window System 89
3.7.6. Fresco 92
3.7.7. Photon microgui 92
3.7.8. VNC 94

4. Descripción de la arquitectura **95**
4.1. Requisitos de diseño 95
4.2. Planteamiento básico 97
4.3. Arquitectura propuesta para el sistema 100

4.3.1. Funcionamiento global del sistema 102
4.4. Representación del estado de los widgets o elementos 103
4.4.1. Ficheros vs. directorios 105
4.4.2. Representación del tipo 107
4.4.3. Representación del estado 108
4.5. Relaciones entre widgets 108
4.5.1. Relación de asociación, A @ B 110
4.5.2. Pertenencia, A ∈ B 111
4.5.3. Relaciones espaciales 114
4.6. Agrupación y codificación de relaciones 114
4.7. Elección del conjunto mínimo de elementos 116
4.7.1. Botón 118
4.7.2. Imagen 119
4.7.3. Barra de desplazamiento 119
4.7.4. Panel 119
4.7.5. Gráficos vectoriales escalables 121
4.8. Contenedores 122
4.9. Codificación de los eventos 123
4.9.1. Árbol de eventos 123
4.9.2. Tipos de eventos: modificación e interacción 126
4.10. Interfaz de programación de los programas cliente 129

5. Descripción de la implementación 131
5.1. 9P: Protocolo de Plan 9 para exportar
árboles de ficheros y su librería 131
5.2. Librería de hilos de Plan 9 134
5.3. Control, librería de widgets de Plan 9 135
5.4. Evolución de los tres prototipos 135
5.5. Concurrencia, sistema de ficheros, interfaz de usuario 138
5.6. Protocolo de mensajes en los pipes 140
5.7. Sistema de ficheros 142

5.8. Utilización de control 143
5.9. Minilenguaje de uso del ratón 144
5.9.1. Primer prototipo 145

6. Evaluación de la arquitectura 153
6.1. Heterogeneidad, diferencia de capacidades y portabilidad 153
6.2. Automatización de tareas y reificación de la interfaz 155
6.3. Adaptabilidad 157
6.4. Protección 158
6.5. Mantenimiento transparente de vistas 159
6.6. Recolección de basura y protección 160
6.6.1. Recolección de basura 160
6.6.2. Protección 162
6.7. Rendimiento 162

7. Conclusiones y trabajo futuro 165
7.1. Limitaciones de la solución 166
7.2. Contribuciones 167
7.3. Trabajo Futuro 168

Bibliografía 171

Índice alfabético 185

Índice de figuras

1.1. Esquema de una arquitectura de interfaces de usuario clásica . 23

2.1. Migración a base de vistas 36

3.1. Reparto de la pantalla de un sistema de ventanas 42
3.2. Interfaz de linea de comando 46
3.3. Componentes de MVC y sus relaciones 69
3.4. Componentes de MVP y sus relaciones 71
3.5. Relación entre MVC y MVP 72
3.6. Arquitectura de Protium 78
3.7. Ejemplo de uso de Ajax 79
3.8. Ejemplo de aplicación usando XML11 81
3.9. Arquitectura de X Window System 90
3.10. Sistema de capas de Photon 93

4.1. Arquitectura con sistema de ficheros multiplexor común 97
4.2. Arquitectura con sistema de ficheros multiplexor por cliente . . 98
4.3. Arquitectura con librería multiplexora 98
4.4. Arquitectura con sistema de ficheros p2p 99
4.5. Arquitectura con sistema de ficheros asociado a múltiples vistas 99
4.6. Arquitectura propuesta, Oni 101
4.7. Funcionamiento global del sistema. 104
4.8. Tipos de relaciones entre elementos 109

4.9. Metaelementos fila y columna; dos relaciones de contigüidad . . 115
4.10. Dos tipos diferentes de botones 118
4.11. Barra de desplazamiento 120
4.12. Cada trazo se corresponde con una representación textual . . . 121
4.13. Estructura plana versus jerarquía de eventos 124
4.14. Ejemplo de árboles de eventos y de estado 125
4.15. Inyección de un evento en un fichero **events** 126
4.16. Procesado de eventos de modificación y de interacción. 127

5.1. 9P como RPCs y como llamadas a función 134
5.2. Widgets de control y uso de controlwire() 136
5.3. Estructura de la implementación 140
5.4. Threads existentes en la implementación 141
5.5. Protocolo de los pipes existentes en la implementación 142
5.6. Estructura del procesado de V1 146
5.7. Estructura del segundo procesado de V2 151

6.1. Recolección de basura desde ambos puntos de vista 161

7.1. Librería de adaptación para programas escritos para GTK . . . 168

Capítulo 1

Introducción

1.1. Computación ubicua

Los computadores están desapareciendo de nuestra atención al hacerse poco a poco más presentes y ubicuos en el entorno cotidiano.

Algunos visionarios como Weiser [153] y otros [151] [155] hablan de un futuro próximo en el que los ordenadores se difuminarán y desaparecerán al convertirse en parte del entorno. Este futuro está cada vez más cerca. Cada vez hay más ordenadores integrados en la vida cotidiana y esta tendencia crece debido a la ley de Moore [120], que permite una mayor miniaturización de los mismos y a la disminución de su coste debida a la producción en masa. En los últimos años se han integrado ordenadores en teléfonos móviles [127], dispositivos portátiles [122], electrodomésticos [67] y coches [141].

La idea de que los ordenadores dejen de ser objetos distintos identificables por el usuario y pasen a estar integrados en el entorno, recibe el nombre de *computación ubicua* [152]. Para que la computación ubicua se haga realidad, es necesaria la miniaturización de los ordenadores y su integración en el medio, es decir, que haya una gran cantidad de ordenadores *invisibles*, al desaparecer de nuestra atención.

En la actualidad, el entorno está formado por muchos dispositivos separados

y sin ningún tipo de integración. Estos dispositivos tiene diferentes capacidades de comunicación y se utilizan en diferentes circunstancias, con la atención del usuario normalmente dividida entre varias tareas. El tipo de interacción casual de la que nos habla Weiser está todavía muy lejos.

En particular, las interfaces de usuario necesitan un cambio sustancial y cualitativo para poder hacer la computación ubicua realidad. A pesar de los grandes cambios que hay en el tipo de dispositivos que se encuentran a nuestro alrededor, la mayor parte de las interfaces de usuario continuan basándose en el modelo clásico de interacción basada en una pantalla, teclado y puntero, que requiere la atención constante del usuario. Para que los computadores desaparezcan, estas interfaces deben cambiar, ser capaces de hacerse parte del entorno, adaptarse a diferentes modos de interacción con el usuario, automatizarse basándose en el contexto y en general desaparecer de la atención del usuario, que debería usarlas de forma casual y refleja. **Esto no es posible si seguimos utilizando interfaces de usuario y arquitecturas de interfaces de usuario diseñadas para el escritorio de un ordenador de sobremesa**.

Las nuevas interfaces de usuario adaptadas a entornos ubicuos deben ser capaces de reubicarse y en general de adaptarse a las necesidades de interacción de los usuarios y no al revés [55]. Esto significa aceptar entradas de voz o más en general, interacciones multimodales, en algunos casos mediante la manipulación de objetos cotidianos. En general, se debe establecer una separación real entre aplicación e interfaz que permita adaptar esta última de forma tan flexible como requiera el usuario. El tipo de interacción a la que nos referimos, puede incluir, por ejemplo, la programación de televisión en casa mediante la voz, la utilización del mando de volumen del móvil para manejar el aire acondicionado o incluso la utilización de gestos manuales para desconectar el equipo de música.

La integración de este tipo de interacción casual hace que los objetos cotidianos parezcan estar dotados de una cierta inteligencia. Por ello, otro nombre que recibe la computación ubicua es "cosas que piensan" ("things that think") [56]. Otro sinónimo de computación ubicua es "computación pervasiva" ("pervasive

computing") [119].

La computación ubicua está también estrechamente relacionada con la inteligencia ambiental [20] o los espacios inteligentes [40]. En este caso, en lugar de centrarse en dispositivos separados adaptándose y colaborando, se contempla el problema desde el punto de vista de la *infraestructura*, es decir, servicios centrales en un entorno controlado y cerrado que posee unos ciertos servicios que se ofrecen al conjunto. En ambos casos se intenta integrar de forma invisible computadoras en el entorno, pero en *inteligencia ambiental* [45] o *AmI* tambien denominada *espacios inteligentes*, este entorno está controlado y cerrado. La computación ubicua intenta hacer lo mismo pero en un entorno abierto en el que los dispositivos sueltos colaboran entre sí. En ambos casos, las interfaces de usuario deben adaptarse al sistema. El diseño de los sistemas es más sencillo en el caso de los espacios inteligentes, ya que siempre se puede suponer que hay presente una cierta infraestructura que en nuestro caso, se corresponde con un servicio de interfaces de usuario. Sin embargo, en computación ubicua, los problemas que se tratan de resolver son los mismos, aunque quizás con una mayor dinamicidad del entorno, lo que los complica aún más.

Los interfaces atentos [147], están también muy relacionados con la computación ubicua. La idea de las interfaces atentas, es que la atención del usuario es un recurso limitado y hay que gestionarlo cuidadosamente. Esto es particularmente cierto en entornos de computación ubicua en los que una miríada de dispositivos rodean al usuario y compiten por su atención. La tecnología diseñada en estas líneas recibe tambien el nombre de tecnología tranquila [154] al intentar no molestar al usuario y dejarle tranquilo en la medida de lo posible.

Un mal diseño en las interfaces de usuario en estos entornos puede claramente llevar al usuario a un estado de estrés constante, al requerir su atención de forma continuada y que configure todos los dispositivos presentes. En estas líneas, los sistemas autonómicos, autoconfigurables y en base a agentes, intentan descargar del usuario parte de las tareas de mantener, configurar y manejar los dispositivos que le rodean. A pesar de ello, siempre será necesaria cierta comunicación del

usuario con el sistema, y esta comunicación deberá adaptarse al usuario.

1.2. Plan B

En el "Laboratorio de sistemas del GSyC", **ls** [42] , hemos construido un sistema operativo, **Plan B** [27] que intenta responder a los nuevos retos y problemas que nos impone la computación ubicua. Plan B es un descendiente directo de **Plan 9** [103], un sistema operativo de los autores originales de Unix [74] que uniformiza el acceso a todos sus recursos exportándolos mediante sistemas de ficheros. Esto significa que cada recurso se encuentra representado por un árbol de ficheros sintéticos. Estos ficheros no se encuentran respaldados en disco y la realización de operaciones sobre ellos es en realidad una forma de interacción con el sistema. La creación de un fichero en el árbol que representa la red, por ejemplo, puede crear una conexión de red.

Estos ficheros se exportan a la red mediante un protocolo para exportar ficheros a la red llamado 9P [16]. Un árbol de ficheros se puede añadir a otro, de la misma forma que en Unix, mediante una operación de montaje. Existe, pues, un árbol global en el que se encuentran todos los sistemas de ficheros. A diferencia que en Unix, este árbol puede ser diferente para cada proceso y recibe el nombre de *espacio de nombres*. Cada proceso tiene, por tanto, una visión diferente del sistema representada en un árbol de ficheros se denomina espacio privado de nombres [104]. Esto permite adaptar y configurar el sistema de forma flexible y transparente a la red. Por ejemplo, un proceso puede montar el sistema de ficheros que representa el puerto serie de otro ordenador en su espacio de nombres. Para este proceso, que interactúa de igual forma con el puerto serie sea local o remoto, es transparente que está utilizando el dispositivo de otra máquina. La interacción con este dispositivo se realiza en ambos casos a través de operaciones de ficheros, **open, read, write, close, seek**, etc. El protocolo que cruza la red entre ambas máquinas es 9P.

Plan B extiende esta aproximación y la adapta a los nuevos requisitos pro-

puestos por la computación ubicua basándose en tres estrategias:

- Los recursos exportados son de más alto nivel. Esto permite su adaptación a los diferentes interfaces y dispositivos. Por ejemplo, mientras que en Plan 9, el sistema de ventanas, rio [79] exporta rectángulos de video, lo que impide que se pueda fácilmente cambiar la resolución de la pantalla o cambiar entre una interfaz de voz y una de texto, en Plan B, se exportan *widgets*. En su significado original se define widget como el elemento atómico de interacción con una interfaz gráfica. En nuestro caso, los widgets son elementos de algo más alto nivel, que representan un tipo de interacción abstracta, que dependiendo de las necesidades, puede ser un botón o un comando de voz, por ejemplo.

- Los recursos, representados mediante árboles de ficheros, se pueden anunciar a la red mediante el uso de restricciones, que son descripciones estructuradas de los mismos. Por ejemplo, si lo que se anuncia a la red es un dispositivo de audio, este se anuncia con restricciones describiendo su localización, su dueño y en general, todos los datos que pudiesen ser necesarios para identificar el dispositivo. Un recurso anunciado a la red recibe el nombre de **volumen** y es un sistema de ficheros. Al igual que en Plan 9 un volumen puede montarse, es decir, añadirse en el árbol de ficheros local. Los volúmenes tambien se pueden montar en base a restricciones. Esto significa que no se monta un volumen concreto, sino que se especifica una descripción del mismo en base a sus características. Cuando aparece un anuncio de volumen que encaja con la descripción, éste se monta. Igual que en Plan 9, esto se hace para cada proceso, que tiene su espacio de nombres privado. Esto permite que nuevos recursos aparezcan, se reconozcan y se utilicen de forma automática. Continuando con el ejemplo del dispositivo de audio; el programa de mensajería del usuario, podría montar en su espacio de nombres, todos los dispositivos de audio que se encontrasen en la misma localización del usuario y para los que éste tenga permiso. De esta

forma, cuando fuese necesario transmitirle un mensaje al usuario, este se podría escuchar a través del sistema de audio más cercano a él.

- Las aplicaciones intentan evitar mantener conexiones abiertas continuamente. Esto permite que se dejen de utilizar recursos que han desaparecido y en general mejora la adaptabilidad del sistema a un entorno dinámico en el que los recursos aparecen y desaparecen. En el ejemplo anterior del sistema de audio, el programa de mensajería abriría cada vez el fichero representando el dispositivo de audio. De esta forma, si el fichero ha cambiado, el sistema se adapta automáticamente.

Oni es la arquitectura de interfaces de usuario de Plan B y utiliza estas estrategias en su diseño e implementación. Los recursos exportados son widgets, elementos de un alto nivel de abstracción, que no representan en realidad a widgets concretos sino a formas y necesidades de interacción, representando una clase de equivalencia entre, por ejemplo, un botón, un comando de voz, un gesto y en general cualquier interacción que requiera del usuario una entrada booleana. Esto permite que la interfaz de usuario se adapte y cambie entre estos tipos diferentes de entradas.

La interfaz de usuario en nuestra arquitectura es un servicio, representado mediante un árbol de ficheros que. Como tal, se anuncia a la red de forma similar a otros servicios de Plan B, mediante restricciones que describen sus propiedades, como puede ser la modalidad.

1.3. Interfaces de usuario

Una interfaz de usuario [13] es:

> "La parte de un programa informático que permite el flujo de información entre entre el propio programa y el usuario".

Dicho de otra forma, es la parte de un programa que se ocupa de interaccionar con el usuario, ya sea para presentarle información o para que el usuario le

presente información al programa. En realidad esta definición tradicional es algo restrictiva, ya que la interfaz de usuario no tiene por qué formar parte del programa, sino que puede ser un servicio del sistema u otro programa encargado de interactuar con el usuario y presentar los resultados de esta interacción a un programa cliente y viceversa.

Una forma tradicional de construir interfaces de usuario (o de gestionarlos) las interfaces de usuario, es la división de responsabilidades en: sistema de ventanas, manejador de ventanas, librería del sistema de ventanas y toolkit gráfico. La figura 1.1 ilustra esta división.

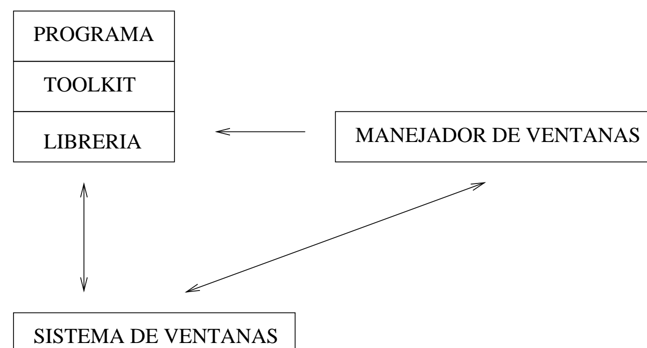

Figura 1.1: *Esquema de una arquitectura de interfaces de usuario clásica*

Esta arquitectura se repite para la mayor parte de los sistemas modernos. Por ello, la utilizaremos de referencia en lo consecutivo. En algunos casos toda la figura se enlaza en un único ejecutable, como en Windows [118] y el sistema se basa en componentes o. En otras arquitecturas, como X Window System [121], el sistema de ventanas y el manejador de ventanas son programas separados que ofrecen servicios distintos.

En general este tipo de arquitecturas maneja elementos de demasiado bajo nivel de abstracción y en muchos casos integrados de forma monolítica en la interfaz de usuario. En X Window System el programa se comunica mediante el protocolo X, que es un protocolo de bajo nivel que dibuja puntos o *píxeles*. Esto impide interponerse entre la aplicación y el servidor para adaptar la interfaz.

En Windows, el sistema de componentes permite una mayor manipulación, pero no existe una separación clara entre interfaz y logica de aplicación que permita interponerse. Además, el sistema es extremadamente complejo.

En muchos sistemas estructurados como en la figura 1.1, hay frameworks, toolkits y en general librerías que ayudan a la construcción de interfaces de usuario.

Estos frameworks o toolkits suelen definir como componentes básicos los *widgets*. Un ejemplo de widget puede ser un botón o un menú. Las interfaces de usuario en estos sistemas suelen desarrollarse mediante la generación automática de código. El programador describe de alguna manera, en muchos casos mediante un editor visual, el aspecto de la interfaz de usuario y una parte de su comportamiento. El editor genera el código necesario para la interfaz utilizando los widgets del toolkit. Este código está escrito de tal forma que cuando llega un evento, es decir, cuando sucede una interacción con el usuario se llama a una función que ha sido registrada previamente por el programador. Este mecanismo se denomina *callback*. El programador escribe la lógica de aplicación y escribe las funciones a las que se va a llamar mediante callbacks.

La arquitectura propuesta, Oni puede usar un toolkit para programar la parte de la interfaz gráfico, es decir, los widgets, pero no es en sí mismo un tookit, ni encaja limpiamente en ninguno de los elementos de la figura. Oni es un servidor de widgets. Se encontraría en donde ahora está el sistema de ventanas, pero contiene en su interior el manejador de ventanas y parte de las librerías. Se ha extraído todo lo que compone la interfaz de usuario y se ha aislado en un servicio separado. La comunicación se realiza a través de un sistema de ficheros y los elementos que se manejan son widgets. En este contexto un widget no es más más que un elemento indivisible de una interfaz de usuario con comportamiento autónomo.

1.3.1. Interfaces de usuario ubicuos

Una interfaz de usuario en un entorno ubicuo tiene una serie de peculiaridades y requisitos especiales que debe cumplir. Estos requisitos están relacionados con la necesidad de adaptarse al usuario, de forma que las aplicaciones salgan de su atención y se integren en el entorno. Para ello es necesario que estas interfaces puedan cambiar de modo, es decir comunicarse mediante voz, texto o el método de interacción que haya disponible, puedan tener varias vistas o interfaces alternativas posibles, que se pueda acceder a ellas a través de la red y en general que se adapten de forma transparente a un entorno heterogéneo y dinámico.

Una propiedad especialmente interesante para estas interfaces es que sean reflexivas, es decir que se puedan manejar componer y descomponer de forma programática. Esto permite adaptarlas de forma flexible mediante programas externos.

La presente tesis consiste en el diseño y validación de una arquitectura novedosa para la construcción de interfaces de usuario orientada a interfaces ubicuos.

Para que una interfaz de usuario pueda cumplir los desafíos que plantea la visión de Weiser, debe cumplir una serie de requisitos que no coinciden con los requisitos de las interfaces de usuario tradicionales. Estos requisitos los plantea tanto la complejidad del entorno distribuido y heterogéneo de ejecución como la necesidad de adaptabilidad y programabilidad de las interfaces.

En el siguiente capítulo, presentamos los problemas que han motivado la presente tesis y los objetivos que se persiguen en ella. A continuación el capítulo 3 , introduce la terminología propia de este campo y describe los diferentes métodos de construcción de interfaces de usuario que existen y han existido en el pasado. Esto nos permite analizar los métodos y mecanismos ya existentes y por qué no cumplen los requisitos presentados. En el capítulo 4, presentamos Oni, nuestra arquitectura de interfaces de usuario para sistemas ubicuos. A continuación, en el capítulo 5 presentamos una implementación de la misma y en el capítulo 6, comprobamos que cumple los requisitos presentados. Finalmente en el capítulo 7 presentamos las conclusiones extraídas del estudio de la problemática de las

interfaces de usuario ubicuos y más en particular del diseño y la implementación de una arquitectura; Oni. También se presentan, en el capítulo 7, las líneas de trabajo futuro que señalan los resultados de esta tesis.

Capítulo 2

Descripción del problema y objetivos

Una interfaz de usuario tiene dos interfaces, una con el usuario y otra con el programa al que representa. La tarea de la HCI [44] o "Human Computer Interaction" es el estudio de la interacción entre el usuario y la interfaz de usuario, estudiando aspectos como la usabilidad y la eficacia de uso. La presente tesis no se ocupa de estos problemas que se mencionarán sólo de forma tangencial.

El problema que nos ocupa es la otra interfaz: la relación entre la interfaz de usuario y el programa. El problema a tratar es el diseño de una arquitectura que permita que esta relación se adapte a los requisitos de un entorno ubicuo, siendo lo suficientemente flexible como para soportar diferentes tipos de interfaces y permitir que se adapten a las necesidades de interacción del usuario.

La resolución satisfactoria de este problema supone también proveer al programador de aplicaciones y/o interfaces de usuario de mecanismos que le permitan diseñarlos y construirlos de forma que se adapten al entorno ubicuo.

Un entorno ubicuo es un entorno que combina dispositivos móviles, portátiles e integrados en él y cuyas principales características [124] son su heterogeneidad y dinamicidad.

Estas dos características son las que plantean un desafío a la hora de di-

señar una arquitectura para interfaces de usuario en entornos ubicuos y tienen como consecuencia una serie de requisitos que debe cumplir para que sea posible utilizarla en estos entornos:

- **Interoperabilidad entre dispositivos heterogéneos y adaptación a diferentes capacidades**. Dispositivos de diferentes capacidades de computación, red y entrada/salida, deben ser capaces de utilizarse entre sí como interfaz de usuario unos de otros y la interfaz de usuario debe ser capaz de adaptarse a las diferentes capacidades de éstos.

- **Reflexión y programabilidad** para adaptarse a la dinamicidad del entorno. Debe ser posible modificar la interfaz de forma programática basándose en el contexto, la aparición y desaparición de recursos y las necesidades de comunicación del usuario.

- **Adaptación modal, separación de lógica y presentación** que permita adaptar la interfaz a las necesidades del usuario y del entorno. El tipo de interacción puede necesitar cambiar entre comandos de voz e interacción mediante una interfaz táctil, por ejemplo, o entre diferentes niveles de detalle según la cercanía del usuario al dispositivo.

- **Un modelo de protección** que se adapte a este nuevo entorno.

- **Mantenimiento transparente de vistas y migración** que permitan adaptar la interfaz a las necesidades del usuario.

En las secciones que siguen analizamos cada uno de estos requisitos por separado:

2.1. Interoperabilidad entre dispositivos heterogéneos y adaptación a diferentes capacidades

La primera barrera a la hora de construir interfaces de usuario adaptables que puedan combinar los diferentes dispositivos que forman el entorno del usuario es la heterogeneidad de estos dispositivos. El hardware y el software sobre el que deben ejecutar las interfaces de usuario ubicuos es extremadamente heterogéneo [107].

Por un lado, el hardware tiene diferentes dispositivos de entrada/salida, tanto en capacidad, por ejemplo, el tamaño de las pantallas, como en modalidad que puede ser audio, vídeo, diferentes tipos de teclados, pantallas táctiles, etc. Además, la capacidad de procesamiento varía desde pequeños dispositivos sin unidad de coma flotante ni MMU[1] [43] y una pequeña memoria flash hasta ordenadores personales de última generación.

Por otro lado, el software también es altamente heterogéneo y varía desde pequeñas librerías de ejecución en reproductores de MP3 a sistemas operativos distribuidos. Incluso aunque nos restrinjamos a ordenadores con unas capacidades mínimas, tendríamos que considerar múltiples sistemas operativos, como Linux [34], MacOS [47], Windows [118] o Plan 9 [103] y lenguajes de programación atados a entornos concretos, por ejemplo, Java [24] sobre algunos teléfonos móviles [117] [58] o Windows, C [75] u Objective C [106] en MacOS, C++ [134] en Symbian [136], C# [66] o C++ para .Net [140] en Windows y Python [146] en muchos frameworks de Linux.

Esta heterogeneidad afecta de forma especialmente negativa a la interoperabilidad. Es complicado conseguir que un dispositivo utilice a otro como interfaz de usuario, debido a la heterogeneidad de software y hardware que utilizan. Sin

[1] La MMU o unidad de gestión de memoria es el hardware del procesador que se encarga de traducir entre direcciones de memoria físicas, las direcciones reales y direcciones de memoria virtual.

embargo, en prácticamente todos los sistemas existen librerías y mecanismos para acceder a sistemas de ficheros en red. Nuestra arquitectura, Oni, aprovecha esta cualidad al ofrecer la interfaz de usuario a través de un sistema de ficheros. De esta forma permite que sistemas heterogéneos se utilicen unos a otros a través de la interfaz de ficheros para mostrar sus interfaces de usuario.

Además de la heterogeneidad de hardware y software otro problema son las limitaciones de la red en sí, que en muchos casos contará con poco ancho de banda, como puede ser una conexión por Bluetooth o gran latencia como sería una conexión ADSL o una conexión GPRS a través del móvil.

Los sistemas que exportan los gráficos a través de una interfaz de bajo nivel, como X Window System [121], rio [79] o VNC [143] acusan problemas cuando se utilizan a través de redes con poco ancho de banda o gran latencia. Para el X Window System existen varios mecanismos paliativos que tratan de soslayar este problema. Uno es LBX [129], que consiste en un proxy en el lado del cliente que comprime las RPCs del protocolo y utiliza extensiones en el lado del servidor para descomprimir estas RPCs. Otra solución es DXPC [148] que es similar pero utilizando proxies tanto en el lado del cliente como en el del servidor para realizar la compresión y descompresión. VNC utiliza varios sistemas de compresión, como puede ser hextile (mecanismo propio de VNC) o incluso MPEG [143].

El problema de estos sistemas es que complican tanto el cliente como el servidor. Además, en algunos casos la latencia que añaden hace el sistema inusable debido a las pobres capacidades de procesamiento del cliente o del servidor. Por eso es preciso que se utilice un lenguaje de comunicación de más alto nivel que al introducir una mayor carga semántica en cada mensaje, utilice menos ancho de banda. Si con un mensaje se puede en lugar de dibujar pixels individuales, crear un botón, se pueden realizar muchas más operaciones con mucho menor tráfico de red.

2.2. Reflexión y programabilidad

Otro reto a la hora de construir una interfaz de usuario es su necesidad de adaptación. La opacidad programática y ausencia total o parcial de reificación y reflexión de las interfaces de usuario actuales obstaculiza en gran medida la adaptabilidad.

Tomemos como ejemplo la arquitectura tradicional que presentábamos en la figura 1.1 en la sección 1.3, por ejemplo el sistema de ventanas X [121], con un manejador de ventanas, sawmill [65] y una librería de widgets como GTK+ [139]. En estas aplicaciones la interfaz de usuario se encuentra incrustada en la aplicación y es difícil acceder a su estructura para controlarla de forma automática.

Si, por ejemplo, se desea acceder a todos los paneles de texto de una aplicación externa para leerlos usando una interfaz de voz, hace falta reprogramar la aplicación o al menos la librería de widgets. Sistemas como MacOS y Windows XP permiten esto, a costa de hacer que los componentes del sistema tengan soporte para ello. Cada nueva capacidad supone modificar o extender todos estos componentes.

Es imposible, también, descomponer la aplicación en partes y utilizar diferentes dispositivos de salida para mostrar cada parte o reagruparlos según convenga en función del entorno en cada momento. Aunque la interfaz de usuario esté formada por componentes independientes autocontenidos, como botones o paneles de texto, no es posible particionarla y controlar los diferentes componentes desde máquinas diferentes.

Un posible ejemplo de esto sería el control de volumen y botones de play y stop de un reproductor de sonido. Al usuario quizás le gustaría poder mover estos controles a la interfaz de usuario del reloj de pulsera para, según se mueve por la habitación poder controlar la reproducción de música. También podría ser interesante tener partes de la interfaz de usuario que sigan al usuario y partes que no.

El software escrito en base a componentes intenta solventar algunos de estos

problemas. Ejemplos de este tipo de aproximación son OLE [35] o DCOM [89] en Windows, Bonobo [110] en Gnome [111] o Kparts [46] en KDE [112]. Los componentes permiten, en teoría, inspeccionar la interfaz de una aplicación en tiempo de ejecución, controlar las aplicaciones remotamente, particionar interfaces de usuario o incrustar una aplicación dentro de otra. En la práctica, los componentes son demasiado pesados o complejos de usar. Bonobo se abandonó en la mayor parte de las aplicaciones por esta razón y en cualquier caso están demasiado atados a un entorno particular, como sucede con DCOM.

Una solución alternativa es la de Morphic [83], que intenta dar algunas respuestas a algunas de estas preguntas. El objetivo de Morphic es más bien exponer al diseñador de interfaces una interfaz particionable y editable desde el propio interfaz de usuario. Para ello define una serie de propiedades. A alto nivel, define **llaneza** (directness) y **viveza** (liveness).

- **Llaneza** significa que se puede acceder a cualquier propiedad del interfaz de usuario a través de la propia interfaz de usuario, es decir una especie de reificación interactiva.

- **Viveza** significa que la interfaz de usuario es una entidad independiente con comportamientos independientes y se puede manipular como tal.

Estas propiedades no son muy importantes para nosotros, pero están ligadas a otras que sí lo son y que también definen los autores de Morphic. Estas propiedades, más simples, pues las anteriores dependen de ellas, son las siguientes:

- **Reificación estructural**. Los elementos se pueden componer en elementos más grandes que se manejan de forma similar a los elementos sueltos.

- **Reificación de posicionado**. El posicionado de elementos es automático y se puede modificar externamente.

- **Comportamiento autónomo**. Los elementos tienen vida, entidad y comportamiento a pesar de encontrarse separados de los programas y de otros elementos.

Las dos primeras propiedades son aspectos importantes de la reificación, que permiten agrupar elementos en compuestos y permiten tratar de forma automática el posicionamiento. La tercera propiedad, comportamiento autónomo es importante para la particionabilidad, así como para la depuración y construcción de interfaces de usuario, para la migración y mantenimiento de vistas de forma sencilla.

Lamentablemente, Morphic tiene otros problemas graves, como la ausencia de protección que impida que un error en un trozo de código tire abajo el sistema. Todos los objetos en Squeak [70] se encuentran en el mismo dominio de protección.

2.3. Adaptación modal. Separación de lógica y presentación

Las diferentes capacidades de los dispositivos y las diferentes situaciones en las que los usuarios interaccionan con ellos hacen necesaria la adaptación modal de las interfaces de usuario. Una interfaz de usuario que requiera interacción gráfica puede pasar a ser de audio o combinar ambos medios, por ejemplo, si el usuario se encuentra conduciendo.

Muy relacionada con este aspecto está la separación entre lógica y presentación. Debido a las diferentes capacidades de los dispositivos de entrada/salida puede ser necesario realizar una adaptación aunque no sea modal. Por ejemplo, cambiar la estructura de menús o mostrar sólo ciertos controles para aprovechar una pantalla pequeña.

Con este fin es importante separar la lógica de la aplicación de su interfaz de usuario. De esta forma se puede cambiar una sin afectar a la otra.

Existen sistemas, como Y Windows [85], proyecto que ahora recibe el nombre de DsY-Windows[11] que desacoplan presentación y lógica de forma similar a la propuesta en esta tesis. Sin embargo, la interfaz que presentan a la red no trata a los recursos de forma lo suficientemente general. Como consecuencia,

a pesar de existir la separación entre presentación y lógica, es difícil establecer propiedades, como la protección, que afecten a los recursos relacionados con la presentación de forma separada e independiente. Además, carecen de la programabilidad necesaria de la que nos provee el hecho de tener una interfaz homogénea. La arquitectura propuesta, Oni, tiene la ventaja añadida frente a estos sistemas de tener la posibilidad de utilizar herramientas ya existentes para la manipulación de ficheros.

2.4. Protección

La protección es un aspecto muy importante, que se ha descuidado mucho en las interfaces de usuario distribuidas, (de los cuales las ubicuas forman parte). En el momento en que las interfaces de usuario se exportan a la red y es posible utilizarlas de forma remota, la protección pasa a ser un requisito indispensable. Sin embargo, no es sencillo construir un modelo de protección que permita compartir interfaces de usuario entre varios usuarios y mantener un control de grano fino en cuanto a que recursos se exporta, cuales no y qué se puede hacer con cada uno.

X Window System, un ejemplo popular y tradicional de sistema capaz de exportar interfaces, tiene a causa de esa capacidad una larga historia de vulnerabilidades.

En sus comienzos utilizaba listas de hosts a los que se permitía acceder, claramente inseguras. Posteriormente se utilizó el sistema de galletas mágicas del MIT ("Magic cookies"), que eran claves de sesión basadas en cadenas aleatorias compartidas entre los clientes y el servidor mediante un fichero. En la actualidad, la práctica más extendida es el uso de un túnel SSH entre cliente y servidor. El sistema actual, considerado seguro, no permite compartir interfaces entre varios usuarios, exportar sólo parte de una interfaz de usuario a otro sistema y no permite, por supuesto, otras políticas de uso más sofisticadas que simplemente la interacción cliente/servidor. Una descripción de los sistemas de autenticación

disponibles para X Windows se puede ver en [48].

Otros sistemas, como el escritorio remoto de Windows XP ("Remote Desktop") [90], o VNC funcionan de forma similar, exportando la interfaz de un cliente a un servidor sin permitir un particionamiento mayor de la interfaz.

Un sistema que permite exportar interfaces de forma más sofisticada es rio [79] en Plan 9, que exporta ficheros representando rectángulos de vídeo y diferentes ficheros de texto representando la consola. Esta estrategia permite un mayor control sobre qué se exporta a la red al ofrecer el modelo de protección de ficheros de Unix, lo que permite compartir y exportar interfaces de usuario. Sin embargo, en rio, el nivel de abstracción y la granularidad de los elementos exportados es insuficiente. Rio exporta ventanas completas. Esto para una aplicación en modo gráfico en Plan 9 significa un rectángulo de vídeo sin reificación de ningún tipo. No se puede exportar parte de la interfaz de usuario de un programa, ni combinarla con otra de otro usuario ni combinar múltiples dispositivos para formar una única interfaz.

2.5. Mantenimiento transparente de vistas y migración

Una consecuencia de la movilidad del usuario y de la necesidad de adaptación de las interfaces a la misma es que introduce a su vez la necesidad de que puedan existir varias copias de estas interfaces en la red. Por ejemplo, podemos desear que el control de volumen se encuentre en el móvil y en la pantalla principal de la habitación simultáneamente. A cada una de estas copias o réplicas del interfaz o de partes de la interfaz las llamaremos *vistas*.

Idealmente, la existencia de distintas vistas debería ser transparente al programador de aplicaciones. Es decir, dichas vistas deberían mantenerse sincronizadas entre sí sin que la lógica de la aplicación intervenga. A esto lo denominaremos *mantenimiento transparente de vistas*.

Si las vistas no se mantienen de forma transparente, en la práctica no es

factible su utilización, pues requieren la conversión de todas las aplicaciones, lo que raramente sucede por el esfuerzo requerido. Además los mismos mecanismos relacionados con la transparencia, como la existencia de una interfaz común a través de las diferentes aplicaciones, unifican estrategias como el recolectado de basura y la reconexión de interfaces, lo que simplifica su programación.

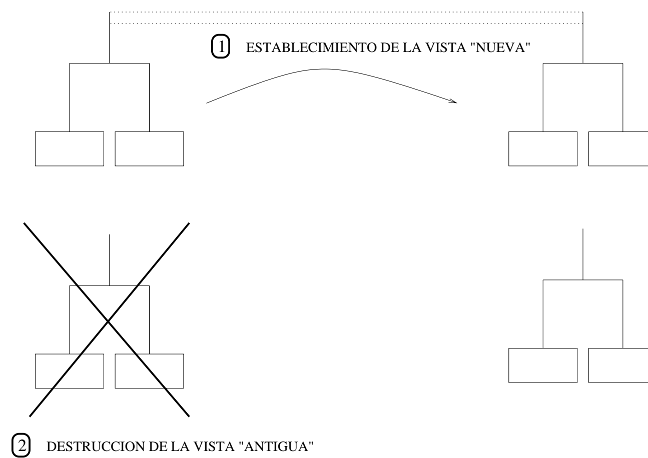

Figura 2.1: *Migración a base de vistas*

Un problema similar es la migración. Desde el punto de vista lógico, la migración de una interfaz de usuario se puede componer de la creación de una nueva vista y la destrucción de la antigua. Esto se puede ver en la figura 2.1. En consecuencia, cualquier consideración que se haga para el mantenimiento transparente de vistas es válida para la migración aunque no al revés. La capacidad de migración no implica la capacidad de mantenimiento simultáneo de varias vistas.

Para X Window System existen ciertas extensiones que reciben el nombre informal de "Display Migration Protocol" [71] y que permiten la migración de ventanas entre displays distintos. Existe una aplicación denominada *teleport* que inicia la migración para aplicaciones que soporten este protocolo. Éstas son pocas, precisamente por la ausencia de transparencia en el soporte al protocolo, ya que la aplicación necesita codificar explícitamente la migración. También

existe soporte en el toolkit GTK+ [139] para utilizar este protocolo, lo que simplifica este problema. Nótese sin embargo, que este sistema utiliza un método de autenticación ad hoc y no permite réplicas en su última versión.

En Windows, la utilización de DCOM permitiría en teoría migrar y mantener réplicas de los objetos distribuidos que forman las interfaces de usuario. DCOM, al igual que CORBA, define estándares de protección y replicación que son extremadamente complicados y pesados. Como consecuencia no hay implementaciones de sistemas que permitan replicación transparente de interfaces de usuario en Windows.

Nuestra propuesta en cambio, tiene un modelo de protección y replicación y además es ligero, pudiéndose implementar en sistemas de poca capacidad.

Mención especial merece *VNC* [143]. VNC permite tanto replicación como migración y replicación de interfaces en Windows, Unix, MacOs y Plan 9. Además, su funcionamiento es independiente de la aplicación. Básicamente, VNC permite exportar la interfaz de vídeo en forma de imagen, el teclado, el ratón y el portapapeles a través de la red. El problema de VNC es que es monolítico y no permite el particionado de aplicaciones.

En algunos sistemas, VNC ni siquiera permite exportar ventanas individualmente, sólo el desktop completo. Además, como ya se ha comentado, VNC utiliza un ancho de banda elevado, pues exporta las imágenes mediante la transmisión a través de la red de cambios o actualizaciones de diferencias en la pantalla a lo largo del tiempo de forma similar a como se haría en un formato de vídeo comprimido, por ejemplo, mpeg. Esto significa que, en conexiones con mucha latencia o bajo ancho de banda, VNC es inusable.

2.6. Conclusión

Los entornos ubicuos plantean nuevos desafíos que las interfaces de usuario tradicionales no pueden abordar.

- La heterogeneidad y dinamicidad del entorno requieren separar la interfaz

de la aplicación y rediseñar la comunicación entre ambos para permitir la adaptación.

- La complejidad del entorno requiere también una mayor programabilidad para permitir la automatización de tareas. Esto se traduce en una necesidad de reificación y reflexión en la interfaz que permita manipular la interfaz de forma automática.

- Deben poder mantenerse vistas de las interfaces de forma transparente.

- El diseño debe ser simple, de forma que se puedan desarrollar diferentes implementaciones del mismo para los dispositivos heterogéneos y que puedan interoperar entre ellas.

- Además, es necesario un modelo de protección que se adapte a este entorno.

Por tanto, es preciso redefinir radicalmente la arquitectura de interfaces de usuario. Proponemos que la interfaz de usuario se separe de la aplicación y que ésta se comunique con aquella a través de un sistema de ficheros con una semántica de alto nivel que se abstraiga de los detalles concretos y permita la adaptación. Al separar la interfaz de usuario de la aplicación y exponer la comunicación mediante un protocolo general, ésta se reifica y se puede controlar de forma automática.

El uso de un modelo bien conocido, como es el de un sistema de ficheros, tiene la ventaja adicional de que hay abundantes estudios previos respecto a cómo distribuirlo y de que ya existen modelos de protección para él. Además, prácticamente todos los sistemas operativos y lenguajes de programación tienen soporte para ficheros, lo que lo hace inmediatamente interoperable si se sigue un diseño cuidadoso.

2.7. Objetivos

El objetivo de esta tesis es, por tanto, **el estudio de la problemática asociada al diseño e implementación de una arquitectura de interfaces de usuario para sistemas ubicuos y la presentación de una arquitectura, Oni, que resuelve esta problemática**. Para ello tiene que cumplir los requisitos presentados de interoperabilidad, reificación, reflexión, separación de lógica y presentación, mantenimiento transparente de vistas y protección.

Capítulo 3

Estado del arte

El estado del arte en interfaces de usuario es tan extenso que necesitaríamos varios tomos para hacer un análisis detallado de todos los toolkits, frameworks y modelos de componentes que existen. Como consecuencia nos vamos a centrar en la interfaz para el programador y en aspectos de ingeniería de sistemas, presentando sólo los ejemplos más significativos o los que desde este punto de vista representan una clase de soluciones.

En este capítulo analizaremos las diferentes formas de construir interfaces de usuario que existen en la actualidad. Dichos diseños arquitectónicos están compuestos de elementos a diferentes niveles de abstracción, en muchos casos con responsabilidades poco claras y solapadas entre ellos.

Estos elementos pueden tomar la forma de:

- Servicios, por ejemplo el que ofrece un sistema de ventanas a las aplicaciones de multiplexar la pantalla haciéndoles creer que tienen toda la pantalla para ellas de forma transparente, como se puede ver en la figura 3.1. Otro ejemplo de servicio es la transparencia de red que ofrece el sistema de ventanas X.

- Toolkits o cajas de herramientas, como GTK [139].

- Frameworks o infraestructuras de programación, como Morphic [83] , la

infraestructura para construir interfaces de usuario de Squeak [70].

- Patrones de diseño, que son convenciones de programación, por ejemplo, MVC [77] que define una serie de entidades y responsabilidades a la hora de construir un interfaz de usuario.

- Programación basada en componentes, como Bonobo [110] en Gnome [111].

- Modelos arquitectónicos, por ejemplo, un sistema de ventanas, un manejador de ventanas y un protocolo para comunicar ambos con la aplicación y las relaciones entre ellos.

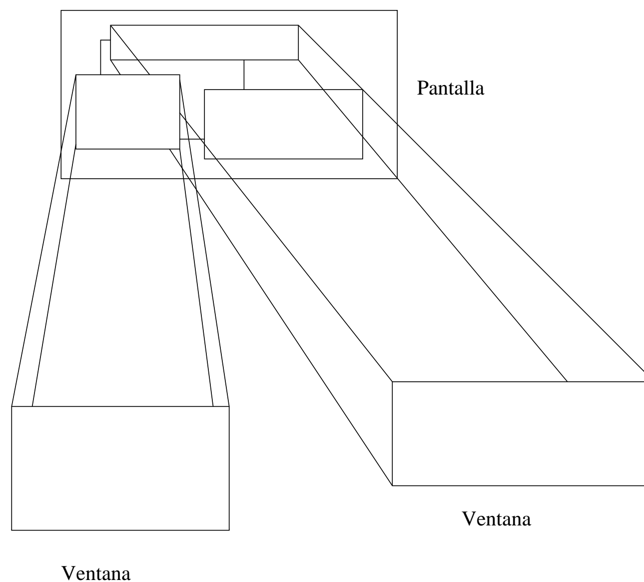

Figura 3.1: *Reparto de la pantalla de un sistema de ventanas*

3.1. Introducción y terminología

Antes de exponer un resumen del estado del arte, describiremos brevemente las diferentes abstracciones, arquitecturas y responsabilidades, introduciendo los

conceptos necesarios para continuar el discurso en el orden adecuado. Posteriormente, en la sección 3.2 realizaremos una taxonomía de los diversos sistemas basándonos en los problemas que abordamos en la presente propuesta y que ya presentamos en la sección 1: interoperabilidad, reflexión y programabilidad, adaptación modal, mantenimiento transparente de vistas y protección .

3.1.1. Mecanismos generales para la distribución

En esta introducción se discutirá la presencia o funcionamiento de algunos mecanismos fundamentales de distribución en las diferentes infraestructuras. A continuación definimos estos mecanismos restringiéndonos a los aspectos que nos interesan a efectos de esta tesis.

Serialización, aplanado/desaplanado o empaquetado/desempaquetado

Serialización es el proceso de empaquetar o codificar el estado de un objeto u otro tipo de dato de forma que se pueda salvar en un medio de almacenamiento o transmitirse a través de la red con el fin de que exista la posibilidad de recrear el estado de dicho objeto a partir de lo almacenado o transmitido. También recibe el nombre de aplanado/desaplanado (marshalling/unmarshalling [137] en inglés).

La codificación del objeto se denomina representación externa.

Nombrado

En los sistemas distribuidos se utilizan esquemas de nombrado [138] para hacer referencia a los elementos distribuidos con transparencia de acceso y de localización. Para ello se asignan identificadores de forma estructurada. Esto recibe el nombre de esquema de nombrado.

Los esquemas de nombrado más comunes tienen una estructura jerárquica. Un ejemplo podrían ser los caminos y los nombres de ficheros y directorios en un sistema de ficheros.

Recolección de basura

Cuando una entidad deja de utilizarse en un sistema distribuido, hay que liberarla y liberar los recursos asociados a ella. Este proceso recibe el nombre de *recolección de basura*.

Para saber si una entidad ha dejado de ser útil, se suele utilizar algún sistema de nombrado o localización para saber si la entidad se encuentra todavía referenciada, es decir si existe alguna forma de acceder a ella. Si la entidad ya no se encuentra referenciada, se recolecta.

3.1.2. Medio, modo y modalidad

En la disciplina de conocimiento que estudia la interacción hombre-computador Mayes [87] distingue entre tres formas de clasificar la interacción:

- *Modalidad* es el nombre que recibe el tipo de comunicación entre el usuario y la interfaz de usuario. La modalidad originalmente se refería al sentido humano utilizado en la interacción, como puede ser el tacto o la vista. Este concepto se ha extendido y hoy día, modalidad se refiere también a la naturaleza de la información utilizada en la comunicación, por ejemplo verbal o espacial.
- *Modo* es el lenguaje concreto utilizado, por ejemplo un menú o un botón.
- *Medio* es una combinación de modo y modalidad, pudiendo referirse a uno de ellos o ambos.

Los expertos en interacción hombre-computador continúan elaborando estos términos que son muy difíciles de definir y están directamente relacionados con la comprensión de sistema cognitivo humano. Para una discusión más detallada sobre este tema, se puede consultar [125].

Modo, medio y modalidad no nos conciernen en esta tesis per se, pues no estamos interesados en usabilidad o formas de interacción. Sin embargo, la arquitectura que proponemos, Oni , trata de permitir cambios de modo y modalidad.

Para ello debe establecerse una separación suficiente entre lógica de aplicación e interfaz de usuario.

3.1.3. Widgets

Un *widget* es el elemento mínimo autocontenido de una interfaz de usuario. comúnmente, los widgets se definen para interfaces de usuario gráficos de tipo clásico en los que la comunicación con el usuario se realiza por medio de una pantalla. Ejemplos típicos de widgets son botones, entradas de texto, o barras de desplazamiento.

En la presente tesis se extiende el concepto de widget en dos sentidos diferentes. Por un lado, se extiende su significado para abarcar elementos no necesariamente relacionados con una interfaz de usuario gráfica. Por otro, los widgets de Oni, representan en realidad un elemento de interacción que puede tener diferentes representaciones modales o incluso diferentes representaciones para un mismo modo. Por ejemplo, un botón representa un cambio de estado. Este cambio se puede realizar mediante un comando de voz, un botón físico, un botón de una interfaz gráfica o un gesto de ratón.

3.1.4. Interfaces de usuario de linea de comando

Los primeros interfaces de usuario, antecesores de las interfaces gráficas, fueron los interfaces de linea de comando o CLIs [9]. En su forma más sencilla, un CLI dibuja una cadena de texto llamada prompt para expresar que está listo para ejecutar un comando. Un glifo, que recibe el nombre de cursor, indica la posición en la que el usuario introduce texto. Mediante una tecla, el usuario indica que desea ejecutar el comando al finalizar la edición. Esto se puede ver en la figura 3.2.

Este tipo de interacción o modo coexiste hoy en día con los interfaces gráficos, pero ya no es el la forma principal de interacción, sino más bien una de las formas de interacción asociada a la modalidad textual.

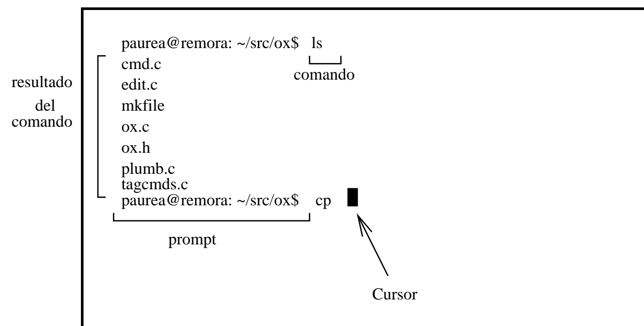

Figura 3.2: *Interfaz de linea de comando*

Otra forma de interacción textual es el intérprete de comandos basado en editor o typescript. Este tipo de interacción es una generalización del CLI que prescinde del prompt y de la ejecución linea a linea. El usuario es libre de seleccionar mediante el ratón o el teclado cualquier región del texto y ejecutarla como un comando. La salida de los comandos, los propios comandos y el editor son tratados de forma similar.

Ambos tipos de interacción, CLI y typescript requieren desde el punto de vista de la interfaz de usuario la edición de texto, tanto por parte del usuario como por parte de las aplicaciones. Requieren también que el usuario pueda expresar que desea ejecutar una cadena de texto. Si los comandos se representan como cadenas de texto, no hay diferencia desde el punto de vista de la lógica de la aplicación entre este tipo de interfaces y otros como pueden ser los de voz o las interfaces gráficas. Es simplemente un cambio de medio.

3.1.5. Sistemas de ventanas

Un sistema de ventanas es un tipo de interfaz de usuario. Utiliza diferentes zonas (normalmente rectangulares) de pantalla, llamadas ventanas, con el fin de que el usuario pueda interaccionar a través de ellas con varias aplicaciones a la vez (ver figura 3.1). Estas ventanas en muchos sistemas de ventanas se pueden solapar, en otros, sin embargo, la pantalla es un mosaico. El concepto

de sistema de ventanas fue desarrollado por XEROX PARC como parte del desarrollo de WIMP [145] (Windows, Icon, Menú, Pointing device) el paradigma de interacción con el usuario más extendido en la actualidad.

Ha habido una gran cantidad de sistemas de ventanas. En Unix, Blit [97], X Window [121] y Fresco [109] son los principales sistemas reseñables. $8\frac{1}{2}$ y rio [101] , son los sistemas de ventanas de Plan 9 [103], basados en la idea de multiplexación transparente [100]. En Qnx, se puede encontrar Photon [115].

La tarea de dar un servicio de interfaces de usuario a clientes ha sido tradicionalmente parte de la responsabilidad del sistema de ventanas. Los sistemas de ventanas construidos como programas separados de la aplicación (en contraposición a los frameworks de los que hablaremos más adelante), como puede ser el sistema de ventanas X [57], son los que han desempeñado hasta ahora la labor de la que se ocupa nuestra arquitectura.

Si vemos el sistema de ventanas como un servicio que se ofrece al resto del sistema, este servicio consta de una serie de responsabilidades o tareas que se puede dividir en tres partes:

- La primera es la *multiplexación de la interfaz.* Con multiplexación de la interfaz, nos referimos al reparto del espacio de pantalla a través de ventanas y el reparto de dispositivos de entrada y los eventos que generan entre aplicaciones, por ejemplo los eventos de teclado y de ratón. Visto desde un punto de vista más general, se trata de la creación de espacios de eventos e interacción de entrada/salida separados. De la misma forma que el sistema operativo reparte el procesador o la memoria entre sus procesos, el sistema de ventanas reparte la entrada/salida entre las aplicaciones.

- La segunda responsabilidad es la implementación de una forma de exportar la interfaz de usuario para poder utilizarlo de *forma remota.* Esto no es siempre posible, aunque suele serlo en mayor o menor medida en los sistemas de ventanas modernos. La posibilidad de utilizar la interfaz de forma remota supone la utilización de algún protocolo que permita controlar elementos gráficos y eventos a través de la red. Conlleva además la

utilización de algún mecanismo de autenticación y posiblemente, de cifrado. Esta tarea se puede llevar o no a cabo de forma transparente.

- La tercera responsabilidad del sistema de ventanas es la de dar un mecanismo de comunicación que permita manipular (crear, modificar y destruir) la interfaz desde la aplicación y la notificación de eventos desde el sistema de ventanas a la aplicación. Esta abstracción debería ser portable y de alto nivel de abstracción para poder sustituir la interfaz y los mecanismos de interacción de éste con el usuario sin necesidad de modificar la aplicación.

El problema de los sistemas de ventanas actuales es que no establecen estas responsabilidades de forma clara, compartiendo parte de ellas con las aplicaciones.

Un problema relacionado con este es la falta de transparencia en la multiplexación de los recursos. Las aplicaciones en muchas ocasiones interaccionan entre ellas realizando tareas que deberían ser parte del sistema de ventanas como puede ser intervenir en la situación de las ventanas directamente.

Otro problema relacionado con éste es que los sistemas de ventanas suelen tener APIs de bajo nivel de abstracción lo que hace necesario para las aplicaciones interaccionar de forma demasiado íntima con el hardware. Como consecuencia, la heterogeneidad de los dispositivos y la adaptación modal de forma independiente de la aplicación se hacen imposibles.

3.1.6. Manejador de ventanas

Un manejador de ventanas es un programa que se interpone entre la aplicación y el sistema de ventanas. Tiene varias responsabilidades:

- *Decoración de la ventana* para añadirle un borde y en general, objetos comunes de manipulación independientes de la aplicación. Esto permite además que el aspecto del escritorio sea uniforme.

- *Situación de la ventana.* El manejador de ventanas se ocupa de mantener una política uniforme de posicionado de las ventanas y en general de permitir la manipulación de la situación y redimensionado de las ventanas.

- *Procesado de eventos* ventana raíz o root window. Esta ventana, normalmente se corresponde con el fondo de pantalla detrás de las ventanas y su control sirve para dibujar y manejar iconos y menús de fondo que controlan el sistema de ventanas.

- En muchos casos *adopción de las ventanas* de alto nivel o de un subconjunto. Esta técnica permite tener un control completo de las ventanas tanto con propósito decorativo como de control y recibe el nombre de "reparenting" o readopción.

- *Manejo de iconos* y otros elementos de interacción externos a las ventanas.

El concepto de manejador de ventanas es parte del reparto de responsabilidades de muchos sistemas de ventanas. Apareció por primera vez de forma explícita en X Window System [121] [57], sistema que trataremos más en detalle más adelante.

3.1.7. Interfaces de usuario telescópicos

Un interfaz de usuario telescópico, o zooming, es un modo de interacción con interfaces gráficos. En esencia, el escritorio es una superficie infinita sobre la que la pantalla ofrece una vista. Esta vista puede desplazarse bidimensionalmente y hacer zoom. Varias aplicaciones populares como Google maps [12], o el escritorio de Mac Os X [4] utilizan este modo de interacción. También hay toolkits como Jazz [29] diseñados para hacer interfaces telescópicos. A efectos de nuestra arquitectura, no hay ninguna diferencia entre múltiples ventanas solapadas y tener las interfaces de la aplicación dispuestas en forma de mosaico o similar en una superficie continua. Este modo de interacción queda confinado en la interfaz de

usuario y aislada de la aplicación. No obstante, una arquitectura general debería permitir la adopción de cualquiera de estos modelos.

3.1.8. Toolkits

Uno de los métodos más antiguos para escribir aplicaciones es el empleo de toolkits o cajas de herramientas. Los toolkits no son más que librerías especializadas con el fin de hacer más sencilla la escritura de interfaces de usuario. Normalmente encapsulan widgets o elementos de interacción gráfica, como pueden ser barras de desplazamiento, botones o paneles de texto.

En muchos casos, a consecuencia del bajo nivel de abstracción de los sistemas de ventanas, la responsabilidad de ofrecer un API uniforme a las aplicaciones recae en los Toolkits. La principal característica que presenta esta aproximación es la uniformidad a todos los niveles, tanto en el aspecto de uso (look and feel) como en su interfaz de programación. Además muchas de estas librerías están portadas a diferentes sistemas.

Sin embargo, estas ventajas tienen una contrapartida importante. Un Toolkit es insuficiente para realizar la tarea encomendada, pues normalmente carece del concepto de servicio.

Al hacer recaer esta responsabilidad sobre el Toolkit, el servicio se integra en la aplicación, lo que hace las aplicaciones más complicadas y no ofrece una interfaz clara para separar la interfaz de usuario de la aplicación con el fin de reificarlo, de adaptar la aplicación o de realizar cualquier otra tarea que requiera modularidad. De otro modo, surgirán los problemas mencionados en la introducción, al no permitirse adaptar la interfaz al usuario, cuya atención es el recurso más escaso del sistema.

La responsabilidad de arbitrar entre las aplicaciones y hacer que estas ofrezcan una interfaz uniforme más allá del maquillaje que puede ofrecer un Toolkit debe recaer en un servicio ofrecido por el sistema.

Cualquier prototipo de la arquitectura propuesta, Oni, se podría implementar utilizando cualquiera de estas librerías, pues por sí mismas lo que ofrecen es

principalmente un mecanismo de abstracción para programar una interfaz de usuario en base a widgets.

3.1.9. Frameworks o infraestructuras de desarrollo

Un framework es una infraestructura organizada con el fin de ayudar al diseño e implementación de programas, en este caso interfaces gráficos. Esta infraestructura normalmente consiste en librerías de programación especializadas y un conjunto de patrones de diseño, convenios o modelos de desarrollo que dan una estructura unificada a las interacciones gráficas con el entorno.

Un framework suele contener también un conjunto de herramientas que implementan estas convenciones o modelos o los usan para asistir al desarrollador en el diseño, implementación y depurado de los programas. En algunos casos, estos frameworks contienen también un conjunto de servicios que se compartirán entre todos los programas construidos mediante esta infraestructura.

Un modelo que se encuentra en muchos casos, subyacente al diseño de la mayoría de los frameworks, es la separación entre lógica de aplicación (también llamada en algunos entornos lógica de negocio), la presentación y el acceso a los datos.

Este modelo es similar al que aparece en el desarrollo web. Intenta separar las tres partes fundamentales de toda aplicación. Por un lado, cómo se presenta la interacción al usuario a través de las interfaces de entrada/salida (posiblemente heterogéneos y multimodales). Por otro, los datos internos que contiene la aplicación (en el lenguaje usado en web, el contenido). Por último, la lógica de la aplicación, es decir, su comportamiento. Cómo se relacionan las tres partes anteriormente descritas es lo que caracteriza cada uno de los frameworks que vamos a describir mas tarde en este capítulo.

Así como los sistemas de ventanas se encontraban en una posición similar a la arquitectura propuesta en cuanto a responsabilidades, los Frameworks ocupan un espacio bastante diferente, dado que no son elementos arquitectónicos separados, sino que son parte de las herramientas utilizadas para construir la

aplicación. En la mayoría de los casos, además, tienen responsabilidades poco definidas más allá de definir componentes o widgets y algún modelo de programación.

Como ejemplos de frameworks populares hoy en día, podemos mencionar el que incluye Microsoft Visual Studio [73] en Windows, y Cocoa [23] para Mac OS.

El problema principal de los frameworks que existen en la actualidad es similar al que presentan los toolkits. Muchos de los servicios y responsabilidades se integran en la aplicación, no estableciéndose una separación clara entre aplicación e interfaz de usuario. Por ejemplo, es imposible interponerse entre wxWidgets o GTK y su aplicación si queremos establecer un servicio de replicación de interfaces. La librería tendría que tener soporte replicación, puesto que la interfaz está incrustada en la aplicación.

De forma similar a lo que sucede en los toolkits, en muchos casos, tampoco se externalizan en forma de servicio actividades que se deben separar de la aplicación. Así se pierde la posibilidad de establecer servicios compartidos entre las aplicaciones que compartan datos entre sí. Por ejemplo, si se quiere desarrollar un programa que trate de determinar qué tarea realiza el usuario en base al contenido de las interfaces de usuario que está utilizando, este servicio es mejor establecerlo de forma separada e independiente ya que centraliza datos de diferentes aplicaciones. Estas aplicaciones van a tener que acceder a datos compartidos que además hay que procesar conjuntamente. Es un caso claro en el que es mucho más sencillo implementar y depurar un servicio independiente que se ocupe de arbitrar el acceso a los datos y su procesamiento. Además existe la posibilidad adicional de no ejecutarlo en sistemas con limitaciones de hardware. Esta flexibilidad es imprescindible en sistemas ubicuos, donde hay una gran heterogeneidad de hardware.

3.1.10. Patrones de diseño

El término patrón de diseño es un término proveniente de arquitectura (en el sentido de Ingeniería Civil) adoptado por la terminología de programación. Se utiliza simplemente para denominar a soluciones arquitectónicas o de diseño en programación que son reutilizables tras un proceso de abstracción para hacerlas más generales. La referencia clásica de patrones de diseño es [54].

Los patrones de diseño son soluciones al nivel de diseño que no se traducen directamente a código. Hay varios patrones de diseño que aparecen comúnmente en el diseño de interfaces de usuario. El más común es MVC [77], otros ejemplos son el wrapper o el observer, ambos detallados en [54].

El concepto de patrón de diseño es posterior a muchas de las abstracciones que referencia. Fue introducido como una forma de establecer una terminología común que permitiese aprender de soluciones ya establecidas para problemas similares a los abordados y que permitiese así mismo unificar conceptos que, en muchos casos, ya se encontraban en la literatura.

3.1.11. Programación basada en componentes

La programación basada en componentes es el siguiente paso que ha dado la programación orientada a objetos. Los componentes son objetos o conjuntos de objetos vistos como un sólo elemento, es decir, como una unidad modular y escritos en base a especificaciones, con el fin de ser reusables y de estructurarse en forma de servicios con interfaces estándar.

Muchos sistemas han utilizado componentes como forma de construcción, como Windows [63] o Gnome [111]. Éstos han tenido su mayor éxito en las interfaces de usuario. Seguramente esto es debido a que las interfaces de usuario están compuestos en su mayor parte por objetos separados con una interfaz clara y es ahí donde la encapsulación en la que se basan los componentes se comporta mejor.

La programación basada en componentes tiene la ventaja de que aísla cada

elemento de los demás y permite su manipulación independiente. Además, en la mayor parte de los casos existen sistemas de serialización y nombrado distribuido de los componentes, que ayudan a cumplir muchos de los requisitos de nuestra arquitectura. Donde suele fallar esta programación es en la seguridad y en la reificación, que paradójicamente suele ser deficiente, normalmente debido al principal problema de todos estos sistemas: su complejidad.

3.2. Taxonomía de las infraestructuras para construir interfaces de usuario

Antes de proceder a exponer los distintos trabajos existentes en este campo, es conveniente reunir todos ellos en tabla sinóptica de referencia. En la tabla 3.1 se califican las diferentes infraestructuras para construir interfaces de usuario en base a los parámetros de diseño expuestos en la introducción: interoperabilidad, reflexión, separación de lógica y presentación, protección, mantenimiento transparente de vistas y migración .

Esta calificación puede ser negativa o positiva. La calificación negativa, se representa con "−", que puede tener dos significados diferentes:

- La infraestructura no considera este parámetro.
- Los mecanismos de los que provee son defectuosos o insuficientes medidos según este parámetro.

La calificación puede ser + o ++ según lo bien estimemos que la infraestructura resuelve el problema.

Esta medida completamente subjetiva, se basa en hecho objetivos, por ejemplo, lo complicado que sea portar el sistema o lo sencillo que sea cambiar la presentación sin afectar a la lógica de la aplicación.

Esta tabla es indicativa y en algunos casos subjetiva, pero en los siguientes epígrafes, al analizar cada uno de estos sistemas, se explicará la razón de

las puntuaciones. Las entradas de la tabla se encuentran dispuestas en orden alfabético.

Algunas de las infraestructuras puntuadas, como Sam [99], en realidad tienen un propósito específico y no son comparables a un servicio completo para el sistema. Sin embargo, lo que nos interesan son *los mecanismos* utilizados y si éstos se pueden generalizar a nuestro caso.

Sistema	Portabilidad	Reificación	Separación lógica pres.	Protección	Vistas
$8\frac{1}{2}$	+	-	-	++	-
Acme	+	+	+	-	+
Ajax	+	++	++	-	+
Awt	+	-	+	-	-
Blit	-	-	++	+	+
DCOM, CORBA	-	++	-	+	+
DPS	-	+	+	-	+
Fresco	-	-	-	+	+
GTK	++	+	+	-	-
Modelos	-	-	++	-	+
Motif	++	-	-	-	-
Mux	+	-	-	++	-
MVC,MVP, Morphic	+	+	+	-	+
NeWS	-	+	+	-	+
NeXTStep	-	+	+	-	+
Photon	+	+	+	-	+
Protium	++	+	+ +	+	++
Qt	++	+	+	-	-
Quartz	-	+	+	-	+
Rio	+	-	-	++	-
Sam	++	+	++	+	++
Scgui	+	+	+	-	+
Swing	++	-	+	-	-
Tcl/Tk	++	+	+	-	-
Tweak	+	+	+	-	+

Cuadro 3.1: *Sistemas de construcción de interfaces de usuario*

Sistema	Portabilidad	Reificación	Separación lógica pres.	Protección	Vistas
Vnc	++	-	+	-	++
wxWidgets	++	-	-	-	-
XML11	+	+	++	-	+
XML	+	+	+	-	+
X Window System	+	-	-	-	+
Oni	++	++	++	++	+

Cuadro 3.2: *Sistemas de construcción de interfaces de usuario(cont.)*

Basándonos en esta tabla y en los requisitos de diseño de Oni, portabilidad a dispositivos heterogéneos, posibilidad de inspección y control de la interfaz, adaptabilidad, protección y mantenimiento transparente de vistas y migración hemos realizado la taxonomía, que presentamos a continuación. Los diferentes sistemas calificados en la tabla se encuentran bajo una sección dedicada al parámetro que resuelven mejor. En los casos en los que resuelven bien más de uno de los requisitos, se mencionará en los epígrafes correspondientes.

3.3. Interoperabilidad entre dispositivos heterogéneos y adaptación a diferentes capacidades

Una de las ventajas de los toolkits y frameworks es la portabilidad que proporcionan. Aunque no establezcan separaciones claras entre aplicación e interfaz de usuario y no sea natural establecer servicios en ellos, son muy útiles como herramienta para portar aplicaciones. Por ello todos los elementos de esta sección son toolkits y frameworks.

3.3.1. Motif

Una de las librerías más antiguas y de mayor uso es Motif. La mayor parte de los sistemas de ventanas modernas implementan un comportamiento similar a Motif. Es el comportamiento mínimo que se requiere hoy en día a un sistema con el paradigma WIMP, ver sección 3.1.5.

Motif surgió simultáneamente a Open Look/OpenWindows. Juntos fueron durante mucho tiempo las dos librerías más importantes para programar interfaces de usuario en X window. OpenWindows, soportado por Sun, prácticamente desapareció y Motif sobrevive junto con otras librerías como uno de los estándares Unix, aunque su uso está en declive. Motif fue desarrollado por la OSF (Open Software Fundation) una organización que originalmente incluía DEC, IBM y

Hewlett-Packard. Está basado en las CUA (Common User Access) de IBM [69], con lo que su comportamiento (look and feel) es similar a Microsoft Windows y a OS/2. Aunque Motif es un estándar de comportamiento, también hay una librería estándar, que es de lo que vamos a tratar. Ésta se encuentra ahora en su versión 2.1.30, pero su uso está en declive, como ya se ha mencionado.

La programación en Motif no es muy diferente de la de otras librerías gráficas, con una jerarquía de widgets, callbacks que se pueden dar de alta para cada widget y un bucle principal que se encarga de recoger eventos. Las principales versiones de Motif permiten enlazar desde C y C++.

No es complicado implementar una versión de nuestra arquitectura, Oni que se comporte como Motif, bastaría con implementarla usando Motif como librería. Tanto el conjunto de widgets elegido por Motif como sus primitivas de colocación y comportamiento se han tenido en cuenta a la hora de diseñar Oni. No obstante nuestra arquitectura es más general, ya que los widgets representan, en general, una interacción multimodal. Motif y la implementación de Oni encajan bien, principalmente porque Oni define eventos e interfaces de forma abstracta y de alto nivel, lo que permite gran variabilidad entre las implementaciones gracias a la separación entre lógica y presentación.

3.3.2. wxWidgets

wxWidgets [10] es un framework en C++ para construir interfaces de usuario que incluye algunas librerías o interfaces de programación extra, como hilos o expresiones regulares, de forma portable. Hay también soporte para enlazarlo con muchos lenguajes de programación.

La principal característica de wxWidgets es que se enlaza con las librerías nativas del sistema, intentando dar a la aplicación el aspecto nativo de cada plataforma. Con este fin, abstrae los objetos concretos de la librería en cada plataforma a alto nivel. Esto no sucede sólo para elementos de la interfaz de usuario, como podrían ser iconos (que en Mac-OS, requieren por ejemplo un número diferente de clicks que en Windows, por ejemplo), sino también para

otros elementos del sistema, como la creación de hilos y el sistema de impresión.

En algunos casos, sin embargo, wxWidgets, se ve obligado a implementar funcionalidad que no da un sistema concreto o incluso si esto no es posible a añadir algunos objetos que sólo funcionan para plataformas concretas.

Esto es debido a que su aproximación de no intentar conseguir la intersección del conjunto de funcionalidades de los diferentes sistemas, sino que intenta aproximar el conjunto completo de funcionalidad en los diferentes sistemas. Esto tiene dos malas consecuencias importantes. La primera es que se termina teniendo que incluir en el framework lo que hace diferente a cada sistema, como la librería de threads, difuminando aún más las responsabilidades del framework. La segunda es que al tener tantos posibles comportamientos e interfaces diferentes, se hace complicado desarrollar aplicaciones consistentes con ellos. Es mejor elegir un lenguaje común de interacción cuando esto sea posible. Ya es suficientemente complicado y heterogéneo el entorno.

Es interesante el uso que hace wxWidgets de “sizer layouts”, un sistema jerárquico de posicionamiento y redimensionado de elementos cuyo objetivo es ser portable. Un elemento devuelve su tamaño mínimo y si tiene la capacidad de crecer en cada una de sus dimensiones. De esta forma se planifica el espacio disponible repartiéndolo de forma adecuada. Esta idea, similar a la de Morph sobre como se organizan los widgets se ha utilizado en la programación de los prototipos de Oni, incluyendo uno posterior al comienzo de esta tesis, Omero.

La versión actual de wxWidgets (Versión 2) soporta todas las versiones de Windows, Unix con Motif y Mac-OS.

wxWidgets se desarrolló originalmente en el Instituto de aplicaciones de inteligencia artificial, en la Universidad de Edimburgo, para uso interno y fue hecho público por primera vez en 1992.

Un aspecto interesante de wxWidgets es como afronta el problema de la portabilidad del aspecto ("look and feel”) de las implementaciones en diferentes sistemas de la librería.

Para abordar este problema habría en general tres aproximaciones diferentes.

1. Intentar implementar en cada sistema el mismo comportamiento, lo que requeriría reescribir la base de código para cada sistema.

2. Utilizar una aproximación de intersección de funcionalidad, utilizando sólo los componentes que se encuentran en todas los sistemas.

3. Utilizar la base resultante de la intersección de funcionalidad de los sistemas extendiéndola, de forma que el resultado se parezca a la implementación nativa salvo por los detalles que no se encuentren en ésta.

Cada una de estas aproximaciones acarrea ciertos problemas tanto de portabilidad, como de integración con el entorno con el que ejecuta la aplicación, por ejemplo, problemas de coherencia del resultado entre sistemas diferentes. Siempre hay un compromiso entre estas fuerzas contrapuestas y la solución no es sencilla.

La aproximación de Oni, similar a la de wxWidgets, consiste en principio en utilizar librerías nativas para implementar los portes de los diferentes interfaces de usuario. Como el diseño de Oni se mantiene en la intersección de la funcionalidad provista en los diferentes sistemas y las diferencias que puedan existir se encuentran ocultas por la interfaz de ficheros, esta aproximación es sencilla de implementar.

Sin embargo, y debido precisamente a que la interfaz servida es uniforme, esta decisión no es vinculante. Se podrían programar implementaciones de Oni con diversos aspectos, sean ya librerías nativas o portes de las librerías de otros sistemas, sin que la aplicación se vea afectada.

3.3.3. Tcl/Tk

Tcl/Tk [94] fue escrito por el Doctor John Ousterhout cuando daba clases en las Universidades de California y Berkeley. Su principal característica es que está asociado a un lenguaje interpretado, Tcl. Tcl es un lenguaje débilmente tipado cuyo uso es, principalmente, el de escribir prototipos de interfaces de

usuario de manera rápida. La idea básica tras Tcl/Tk era intentar permitir algo similar a la programabilidad de los pipes en la linea de comando pero para programar interfaces de usuario. Tcl permite modificar y componer partes ya programadas del código de forma similar a como se componen al ejecutar, por ejemplo "`ls|wc`".

Por lo demás, Tk es similar a Motif, con su mismo Look and Feel. Se encuentra portado a muchos sistemas operativos en una o en otra versión.

Una de las razones del éxito de Tcl/Tk aparte de su capacidad de prototipado rápido es que los scripts de Tcl/Tk o en particular los comandos de Tk se utilizan como **lingua franca** para describir interfaces de usuario, en diferentes lenguajes, como puede ser Perl, Python o Limbo.

En muchos grandes proyectos, se prototipa el interfaz de usuario con Tcl/Tk y luego se embeben los comandos de Tk utilizando alguna librería. Como estos comandos son cadenas de texto, se pueden construir y cambiar dinámicamente y es sencillo manipularlos.

Oni tiene características similares en cierto sentido gracias a la separación entre la interfaz de usuario y los programas. Además, en nuestro caso, el lenguaje de prototipado puede ser cualquier capaz de manipular ficheros, incluyendo la shell, Tcl, o C.

3.3.4. GTK

Tanto GTK [139], como Qt [144] son librerías más modernas, que intentan resolver algunos problemas que no resolvían las librerías más antiguas, como pueden ser Motif o Tk. Ambas están basadas en el uso de callbacks de una forma o de otra para informar de eventos a la aplicación.

GTK, abreviatura de GIMP Toolkit surgió en 1996 de la mano de dos estudiantes, Peter Mattis y Spencer Kimball como una librería para el programa de retocado de imágenes GIMP. Después GTK se escindió de éste, extendiéndose mucho tras la su adopción por parte del proyecto de escritorio para GNU/Linux denominado GNOME. GTK ha sido portada a múltiples sistemas operativos.

GTK está basada en tres librerías, GLib, Pango y ATK. GLib es una librería de bajo nivel que aporta el soporte de threads, el sistema de objetos y tipado para diferentes lenguajes. Pango es la librería para renderizado de texto. ATK es la librería de accesibilidad, con interfaces para lupas, lectores automáticos y lineas Braille. GTK usa señales (una forma de identificar los eventos) y callbacks para el despachado de eventos.

GTK es la librería que se utiliza en Gnome. En Gnome se utiliza CORBA [84] como protocolo de comunicación entre los widgets, con los problemas que esto tiene asociado y de los que hablaremos en 3.7.6.

Por lo demás, lo dicho para Motif y wxWidgets se aplica a este epígrafe y a los que siguen.

3.3.5. Qt

Qt es una librería desarrollada por la compañía Trolltech. En 1996 el proyecto KDE (otro proyecto de escritorio para Linux) popularizó esta librería. Aunque esta librería está escrita en C++, se puede usar desde muchos otros lenguajes. De forma similar a GTK, Qt utiliza señales. No obstante, Qt tiene una forma diferente de especificar los callbacks, haciendo uso de ranuras (slots), para identificar las funciones que pueden esperar como callbacks en la interfaz de widgets. De esta forma se pueden conectar interfaces que sean fuente de eventos (señales) a otros que sean sumideros (ranuras). Así se puede conseguir que haya un menor acoplamiento a pesar del uso de callbacks, al haber una interfaz clara y descrita entre los widgets, tanto de generación de eventos como de consumo de los mismos.

Qt es la librería que se usa en KDE, un escritorio para Linux alternativo a Gnome. En KDE se utilizaba originalmente CORBA, pero debido a la latencia que introducía, a su complejidad y a que no se estaba utilizando para otra cosa que comunicar componentes gráficos y comunicar procesos, CORBA se sustituyó por KPARTS, que es un modelo de componentes. No es pues comparable a Oni, pues no tiene ningún modelo para exponer componentes de Qt a la red de forma

transparente.

3.3.6. AWT

AWT [1] era la librería original de desarrollo de interfaces gráficas que venía incluida en el kit estándar de desarrollo (JDK) de java. Swing es una librería más moderna pensada para sustituir completamente a AWT, aunque si esto sucederá en el futuro, es incierto. En el momento actual conviven ambas.

AWT intenta utilizar los componentes nativos del sistema en el que ejecuta. Por ejemplo en Windows utiliza las Forms de Windows, mientras que en Unix utiliza Motif.

Es interesante que una de las principales razones tras el desarrollo de Swing y los cambios a AWT son los problemas de portabilidad causados por los diferentes comportamientos entre las librerías de widgets nativas y los intentos de solucionarlos para intentar conseguir un comportamiento uniforme con el entorno que no cambie demasiado entre unos sistemas y otros.

La solución ha venido finalmente por desacoplar la capa de presentación del resto de la librería mediante una interfaz de programación lo suficientemente abstracta como para que la presentación pueda cambiar a gusto del usuario, tomando el aspecto que éste desee.

Sobre la relevancia de estos problemas se ha hablado en la sección 3.3.2. Oni es un arquitectura a otro nivel y se considera que será trabajo del que implemente cada interfaz solucionar estos problemas de un modo concreto.

AWT y Swing tienen los problemas comunes de los toolkits que se discutieron en la sección 3.1.8.

3.3.7. Swing

Swing [2], como ya se ha visto es una librería más moderna que pretendía sustituir completamente AWT. En Swing, los componentes están dibujados y controlados por Java, sin utilizar componentes nativos. Swing introduce tam-

bién una arquitectura de dibujado y control, similar a MVC , con el fin de poder separar la representación y el comportamiento de la lógica de la aplicación. Gracias a esto, los widgets de Swing pueden tomar el comportamiento y apariencia "look and feel" de los componentes nativos, mientras que el código se mantiene portable y no se ve afectado.

3.4. Reflexión y programabilidad

Pasamos ahora a discutir los trabajos cuya cualidad principal es la programabilidad de las interfaces o la exposición de los mismos mediante reflexión. Como ya se ha justificado, esto es necesario para automatizar tareas y descargar al usuario, que es el recurso más valioso en un entorno ubicuo.

3.4.1. OLE, ActiveX, COM, DCOM y CORBA

OLE, ActiveX, COM y DCOM [63] son evoluciones de un mismo producto que se han ido refinando y junto con CORBA [149], son modelos para manejar objetos distribuidos que definen serialización, nombrado, recolección de basura distribuida, inspección de interfaces y en general, infraestructuras para tratar con objetos en red de forma transparente.

Aunque la terminología es diferente, y en DCOM se llama *proxy* lo que en el CORBA se llama *stub*, ambos realizan labores similares. La portabilidad se basa en bindings para usarlos con diferentes lenguajes, en el caso de CORBA mediante IDL [93] en el caso de DCOM también mediante un lenguaje de descripción de interfaces, similar al de CORBA pero más sencillo. La utilización de lenguajes de descripción de interfaces supone que utilizar un lenguaje nuevo en el sistema no sea trivial. Se deben definir y establecer correspondencias entre los tipos de datos e interfaces en ese lenguaje y en el lenguaje IDL [93], lo que no es trivial y luego es preciso escribir un compilador de stubs y proxies que haga las traducciones. Estas correspondencias son difíciles de establecer, tediosas de programar y complicadas de depurar al requerir un conocimiento exhaustivo del

significado de los tipos en CORBA, de IDL y del lenguaje en cuestión.

El paso final de este camino lo ha dado Windows convirtiendo su infraestructura DCOM a .Net [140], que es en realidad la misma infraestructura de componentes. De forma similar, tiene bindings para diferentes lenguajes y anotaciones en el código generado pero en el caso de .NET, este código ejecuta sobre una maquina virtual.

Estas infraestructuras se han utilizado, además de otros usos, para definir componentes de la interfaz de usuario, como por ejemplo DCOM en Windows y Bonobo [110], originalmente basado en CORBA en el escritorio Gnome [111]. Definen un sistema de nombrado distribuido y de acceso mediante RPCs a objetos remotos. Permiten, en cierto modo, reificar la interfaz. Cuando se unen al interfaz de usuario, suelen recibir el nombre de componentes. Un componente, en este contexto, es la pieza atómica que compone un elemento de interfaz de usuario con entidad desde el punto de vista programático y suministra una interfaz de acceso al mismo.

La función de toda la infraestructura de componentes la realiza en nuestro caso el sistema de ficheros. En el existe un sistema de nombrado, una forma de serialización (cualquier programa de archivado de ficheros), protección y principalmente reificación, lo que permite la manipulación automática de la interfaz y por tanto la automatización de tareas. Y todo ello de una forma portable sin necesidad de bindings específicos y sin necesidad de un incluir o modificar software.

3.4.2. KPARTS

KPARTS [62] es la infraestructura de componentes de KDE [135]. Es una infraestructura muy sencilla y no tiene muchos de los servicios de CORBA o DCOM. Es sencillamente una serie de interfaces estándar entre objetos, un sistema de linkado dinámico. El posicionamiento se describe mediante ficheros en formato XML. Los problemas ya mencionados para CORBA y DCOM se mantienen en KPARTS, añadiendo la ausencia de algunos de los servicios de aquellos.

Por otra parte, KPARTS es más sencillo y resulta más fácil de utilizar.

3.5. Adaptación modal. Separación entre lógica y presentación

En esta sección describimos sistemas o paradigmas que establecen algún tipo de separación entre lógica y presentación. Esta separación permite la adaptación de la interfaz y el mantenimiento transparente de vistas.

3.5.1. Blit

El Blit [97] fue un terminal gráfico programable basado en representación de bitmaps en pantalla diseñado en Bell Labs en 1983 por Rob Pike y Bart Locanthi. Fue uno de los primeros, si no el primer ejemplo de cómo la multiprogramación en Unix combinada con una interfaz gráfico con ventanas, es decir, multiplexado en diferentes ventanas podía resultar mucho más productivo y cómodo al permitir al usuario trabajar con varias tareas a la vez de forma sencilla.

Los terminales eran dispositivos pequeños que ejecutaban sólo la interfaz de usuario y que se conectaban a una máquina con mayor capacidad de cómputo mediante una línea serie. El dispositivo era muy barato y tenía un procesador MC68000 de Motorola y una cantidad exigua de RAM (256K) y de ROM (24K) y no tenía hardware especial para ocuparse de los gráficos. Ejecutaba un pequeño sistema operativo multitarea integrado con el sistema de ventanas.

Los programas estaban compuestos de dos partes, una de las cuales se descargaba en el terminal y se ocupaba de manejar la parte de la interfaz de usuario.

Por ejemplo, un editor tenía dos subprogramas, la interfaz de usuario y la parte que se ocupaba de los ficheros, es decir, el editor propiamente dicho. El subprograma que se ocupaba de la interfaz de usuario, sólo gestionaba las estructuras de datos (replicadas en el terminal y en la máquina Unix) en las que

se guardaba el texto a representar en pantalla. Este subprograma se comunicaba con el editor propiamente dicho mediante un protocolo sencillo de mensajes dedicado a borrar cadenas e insertar cadenas en los ficheros.

El subprograma de la interfaz de usuario que se descargaba en el Blit se desarrollaba mediante el uso de librerías que hacían uso de objetos geométricos abstractos, pero que, sin embargo, estaban muy atadas al tipo concreto de dispositivo, es decir, el subprograma de interfaz de usuario era portable salvo por las librerías encargadas del manejo gráfico.

El Blit utiliza una aproximación similar a la que seguimos nosotros en nuestra arquitectura. Separa y desacopla interfaz de usuario y programa. Sin embargo, de forma similar a Protium [158] que describiremos más adelante y que es un sucesor de Blit, el protocolo de comunicación entre la interfaz y la aplicación es poco general y cambia para cada aplicación. Esto impide el uso de herramientas generales con él y no provee mecanismos de protección, seguridad y nombrado que se obtienen al usar un sistema de ficheros.

Sin embargo, ya en el Blit se observaba un beneficio importante derivado de construir la interfaz de usuario separada de la aplicación y comunicada mediante un protocolo de alto nivel, la optimización en el uso de la red. Las interfaces se podían conectar con los editores a través de líneas serie, cuya velocidad no es comparable a la de una conexión de red moderna.

3.5.2. MVC

MVC [78] [77] o *Model-View-Controller*, es un patrón de diseño, es decir una forma de organizar programas que aparece en varios frameworks para construir interfaces de usuario de entre los cuales el más importante es Smalltalk-80 [60], en cuyo framework apareció por vez primera tras su invención en Xerox Parc en los años 70. Otro sistema que utilizó este patrón de diseño es Oberon [116].

Posiblemente MVC es el primer modelo que se inventó que describe el desarrollo de interfaces de usuario en términos de responsabilidades con los componentes separados y comunicándose por medio de un protocolo.

En MVC las aplicaciones están divididas en tres componentes que separan eventos, la presentación y el procesado, como se puede ver en la figura 3.3. Aunque es un modelo similar no se corresponde uno a uno con el que presentábamos en la introducción.

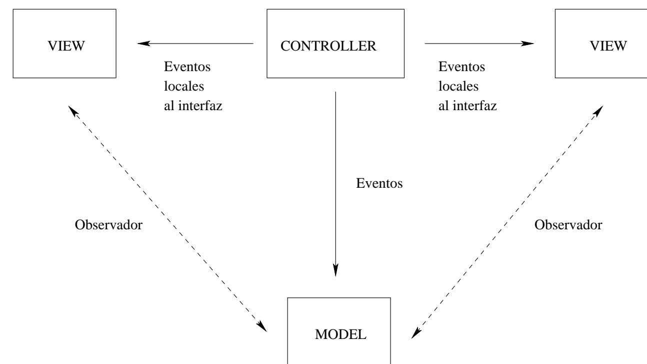

Figura 3.3: *Componentes de MVC y sus relaciones*

- La M, (*Model* en inglés) se ocupa de la lógica de la aplicación, es decir, de la parte de la aplicación que no tiene que ver con la interfaz de usuario. También confina los datos que se están representando. Es el núcleo de la aplicación si no se tiene en cuenta la interfaz de usuario.
- La V, (*View* en inglés) es la vista, es decir, la parte que se ocupa de la presentación. Su único papel es presentar los tipos de datos que se le suministran en pantalla y manejar la interacción del usuario con éste.
- La C, (*Controller* en inglés) representa las operaciones que cambian y controlan la aplicación. Por ejemplo, podría corresponderse con el procesado de ratón y teclado. Es, principalmente, un bucle de eventos, es decir un bucle que recoge eventos de los dispositivos de entrada que proporciona su *View* y cambia los datos del *Model*.

View y *Controller* vienen por parejas y están muy acoplados entre sí, porque hay eventos, como puede ser cambiar la zona activa de la ventana que afectan

directamente al *View* sin cambiar el *Model*, indicados en la figura 3.3 como "Eventos locales al interfaz". Esto sucede más cuanto mayor es la separación entre presentación y modelo.

La relación más interesante es la que existe entre el *View* y el *Model*, que es un patrón observador [54]. Esto significa que los *View*s se añaden a un registro en el Model. Cuando cambia el *Model* (por interacción con el *Controller* o por razones internas), se avisa a los *View*s que como consecuencia realizan una RPC para recoger los datos. Este mecanismo desacopla *View*s de *Model*s, permitiendo, por ejemplo, la existencia de varias vistas independientes sin necesidad de modificar en lo más mínimo el modelo. De esta forma, el *Model* no sabe ni necesita saber nada de la existencia de uno o mas de un *View*s.

Oni utiliza un mecanismo similar al MVC para avisar a sus vistas (diferentes interfaces de usuario) de un cambio en la interfaz. Entre la aplicación y las interfaces se interpone un multiplexor que se encarga de actualizar las diferentes vistas, lo que no es más que repetir las mismas operaciones sobre un conjunto de ficheros. El registro que haría un MVC de una nueva vista aquí se realiza anunciando y montando un sistema de ficheros definido por Oni. Las actualizaciones se hacen en sentido modelo vista en lugar de al revés, lo que evita la RPC de actualización innecesaria. Por otro lado, en nuestro caso es posible que la vista tome mas datos de los necesarios del modelo, que no sabe, por ejemplo, que éstos se encuentran ocultos en esa vista en particular.

Apple ha utilizado a lo largo de su historia varios frameworks, los más recientes basados en MVC.

En los diferentes sistemas operativos de Apple las interfaces de usuario comenzaron con *mprove* [91] la interfaz de usuario de Lisa [156]. Mas adelante apareció el Macintosh que ha tenido un gran número de modelos y que ha ido evolucionando hasta su forma moderna, mezclándose con el software de la empresa NeXT. La última versión del sistema operativo MacOS X utiliza Cocoa una evolución de las librerías de NeXT. Cocoa es un framework orientado a objetos construido en Objective-C que utiliza MVC. Cocoa está basado en Quartz,

del que se hablará más adelante.

3.5.3. MVP

El modelo MVP [22] (*Model-View-Presenter*) es una evolución del MVC diseñada para resolver algunos problemas de este modelo. Apareció por primera vez en el framework para java y C++ de Taligent [157], una subsidiaria de IBM. Después fue adoptado por otros, como Dolphin, un entorno de desarrollo de Smalltalk creado por Intuitive Systems Ltd. y que ahora mantiene la compañía Object Arts.

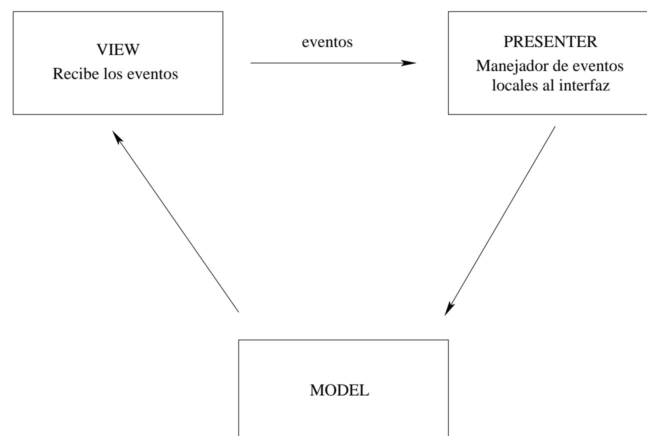

Figura 3.4: *Componentes de MVP y sus relaciones*

En MVP el *Controller* está empotrado dentro del *View*. La P, (*Presenter* en inglés) representa la parte de la lógica de aplicación que se refiere al comportamiento de la interfaz y que estaba estrechamente ligada al *View* en MVC.

En MVC la separación que se producía como consecuencia del uso del patrón observador hacía que la lógica de la interfaz no pudiera comunicarse adecuadamente con la vista y muchas veces, se saltaba por encima de la interfaz, perdiendo todos los beneficios de utilizar MVC.

Ahora, ambas se enlazan juntas y en *Model* sólo queda la parte realmente asociada a la lógica de la aplicación. Dicho de otra forma, el *Presenter* actúa

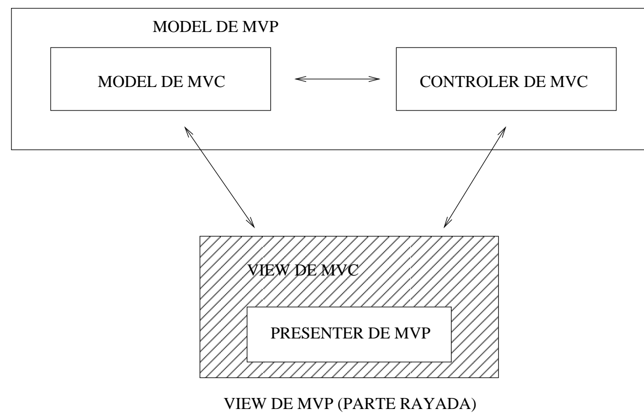

Figura 3.5: *Relación entre MVC y MVP*

como un manejador de eventos locales al interfaz y delega el resto al *Model*. Esta estructura se puede ver en la figura 3.4. La relación entre MVC y MVP se puede ver en la figura 3.5.

En nuestra arquitectura, Oni, la parte de la interfaz que está muy acoplada con la aplicación se encuentra en el interior de los widgets, con lo que el problema que resuelve el MVP sencillamente no existe. El interfaz se encuentra completamente separado de la aplicación y sólo le comunica lo que no es local al interfaz de usuario. La única excepción es la lógica asociada a conectar widgets entre sí que en algunos casos, podría separarse completamente de la lógica de aplicación y que en nuestra arquitectura está incluida dentro de la lógica de la aplicación. Sin embargo, una buena elección de los widgets, como por ejemplo asociar la barra de desplazamiento al widget de texto, minimiza estos casos.

3.5.4. Morphic

Morphic [83] es un framework para la realización de interfaces de usuarios que surgió en Self, un entorno de programación de Sun basado en un lenguaje homónimo. Self es un lenguaje basado en prototipos de objetos cuya principal característica es que los objetos pueden heredar el estado y cambiar su herencia dinámicamente. Estas características se basan principalmente en el uso del

mecanismo de delegación. Morphic forma hoy parte de Squeak [70], un entorno completo de Smalltalk [60] que lo utiliza como paradigma para la construcción de la interfaz de usuario.

Morphic está basado en elementos llamados *morph*. Un morph es un componente de la interfaz de usuario manipulable de forma directa por el usuario. Cualquier morph actúa además como contenedor de otros morphs, llamados *submorphs*. Los morphs son entidades activas y se planifican de forma similar a los procesos en un sistema operativo, recibiendo rodajas de tiempo periódicamente.

Cada morph tiene una representación visual que el usuario puede manipular directamente. Cada morph puede además:

- Realizar acciones en respuesta a entradas del usuario, como cambiar el estado de un botón.
- Realizar una acción cuando un morph es dejado dentro de otro morph, como cuando se arrastra un morph a la papelera.
- Realizar una acción a intervalos regulares, como parpadear.
- Controlar el tamaño y localización de los submorphs, por ejemplo, ponerlos en un grid y darles a todos el mismo tamaño.

Los autores de Morphic definen dos propiedades que definimos en la introducción, llaneza y viveza, que son el objetivo de su diseño. La llaneza es simplemente exponer la estructura de la interfaz a través de la interfaz de usuario. La viveza significa que los componentes de la interfaz son activos e independientes.

Gracias a la viveza, Morphic es un sistema jerárquico de elementos manipulables de forma separada (es sencillo, mediante la interfaz de usuario, romper y volver a unir morphs).

Gracias a la llaneza, la interfaz es sencilla y general. Todos los mecanismos se exponen al usuario, que en cualquier momento puede componer interfaces de usuario usando objetos durante la ejecución o descomponer una aplicación en partes más simples.

Estas propiedades se basan a su vez en la reificación estructural y de posicionado y en el comportamiento autónomo de los widgets. Estas tres propiedades son fundamentales en Oni, nuestra arquitectura. En general, son necesarias para cualquier sistema ubicuo que requiera un alto grado de reificación y particionabilidad, parte de los requisitos de la arquitectura propuesta, Oni, que ya expusimos en la introducción.

Algunas otras ideas de Morphic, como la jerarquía de propagación de eventos y de cambios de tamaños se han utilizado también en la implementación de Oni.

La principal diferencia entre Oni y Morphic es la separación que establece Oni de la interfaz de usuario de la aplicación. Esto proporciona un nivel de protección que no existe en Morphic (en general en Smalltalk) en el que cualquier conjunto de objetos puede mandar mensajes a los demás de forma arbitraria, sin restricciones. Esto hace que el sistema sea muy vulnerable ante errores en los programas. El proyecto Islands [59] intenta resolver este problema en general para Smalltalk mediante el uso de objetos representantes "far references" que actúan de intermediarios, aislando las islas entre sí. Los "far references", deciden si delegan en el objeto al que representan la acción, si la rechazan, la replican, etc.

3.5.5. Tweak

Tweak [7] es la sustitución de Morphic para Squeak. Morphic tenía el problema de que no tenía un modelo claro para construir interfaces de usuario como hacía MVC. Aunque Morphic permite construir interfaces de usuario de forma sencilla y mediante manipulación directa, la falta de abstracciones en los diferentes componentes impide que el código se pueda reutilizar. Además, impide la interoperabilidad de unos componentes con otros.

Tweak intenta resolver este problema añadiendo algo similar a MVC para Morphic. Esta arquitectura se llama Players and Costumes [5]. Un player es algo similar a un *model* en MVC y costume es similar a un *view*. La diferencia principal es que son roles o interfaces más que componentes. Tweak está en

desarrollo en este momento y todavía no está claro si será un avance respecto de Morphic o MVC. Tweak se utiliza principalmente en Croquet, [128] antes llamado Tea, un entorno de colaboración distribuido basado en Smalltalk.

3.5.6. Sistemas basados en XML

XML no es más que un metalenguaje de descripción de lenguajes de marcado. Muy recientemente ha habido una explosión en la aparición de sistemas que hacen uso de XML para la descripción de las interfaces de usuario. Esta corriente comenzó con la necesidad de adaptar las páginas web a los diferentes medios de presentación disponibles (diferentes resoluciones, pantallas de teléfonos móviles e interfaces de voz). Apareció entonces la necesidad de utilizar un mecanismo de descripción de los formularios y formatos independiente de la presentación, en la línea iniciada con el lenguaje de descripción para páginas web original, HTML, que permitiese tener una sola descripción del contenido y varias instancias de la página web en WML, HTML, RDF, PostScript y PDF.

Los dos ejemplos principales de programación de interfaces mediante XML son Glade [39] y Visual Studio [73]. Ambos permiten al programador diseñar de forma visual la interfaz, es decir mediante la interacción con representaciones de los widgets y ventanas con su aspecto final y generar una versión XML del mismo y stubs que permitan integrar la lógica de aplicación. La descripción XML se utiliza para adaptar la interfaz a aplicaciones normales a servicios web. La descripción XML es la interfaz abstracta para establecer la separación entre presentación y lógica de aplicación.

Aunque estos sistemas establecen separación de presentación y contenido, no describen por sí mismos como resolver los problemas que planteábamos en la introducción, replicación, seguridad y migración de interfaces sino que precisan de mecanismos externos que realicen estas funciones.

Todos ellos funcionan como una descripción a alto nivel de un documento, sin partes activas y sin definir la arquitectura necesaria. Para añadir replicación, seguridad y migración hace falta algo más que una descripción de la interfaz y

un intérprete, es necesaria una arquitectura y programas.

3.5.7. Sistemas basados en modelos

Ejemplos de sistemas basados en modelos, podrían ser SUPPLE [3], Mastermind [36] o Plastic User Interfaces [6]. Los sistemas basados en modelos parten de la idea de que las interfaces de usuario se deben generar o adaptar de forma automática desde la lógica de la aplicación. Partiendo de un modelo de interacción y de una lógica de la aplicación se genera de forma automática la interfaz de usuario.

Esto resuelve un problema completamente diferente del que trata de resolver la arquitectura propuesta, Oni. El problema que resuelven los sistemas basados en modelos es el del diseño de la aplicación y en particular de la relación con su interfaz de usuario. Con este fin, intentan automatizar el *diseño* en base a unos requisitos. Oni en el diseño de la arquitectura, intenta proveer al que realiza la implementación, incluso aunque sea un programa el que realiza esta implementación, de *mecanismos* mediante los cuales dotar de ciertas propiedades al interfaz de usuario.

Ambas aproximaciones son compatibles. Es más, la uniformidad de la que provee Oni (tanto en interfaz de programación, como en comportamiento), debido a que se convierte en un mecanismo universal, facilita la tarea de automatización en la creación de interfaces. Además, la propiedad fundamental que permite que un programa manipule la interfaz de forma independiente, es la reificación, uno de los requisitos de diseño de Oni.

En algunos casos, como en el caso de SUPPLE, existe una intersección entre la labor que realiza Oni y la que realizaría el sistema basado en modelos.

La idea de SUPPLE es adaptar la interfaz dependiendo de las características del dispositivo y del usuario. SUPPLE realiza una adaptación de la interfaz a dos niveles, el de presentación y el funcional o de comportamiento.

En el nivel de presentación, diferentes componentes se reflejan de forma diferente dependiendo de las capacidades de la interfaz de comunicación con el

usuario.

Un ejemplo de esto es una barra de desplazamiento que se convierte en la selección de un número en una interfaz sin capacidades gráficas. Con el fin de que esta adaptación se pueda realizar, la interfaz de ficheros de Oni es de suficiente alto nivel y laxa como para que la lógica detrás de los componentes no perciba la diferencia en el cambio de presentación. De esto ya se hablará más adelante en el capítulo de diseño.

El otro nivel de adaptación que implementa SUPPLE, se realiza tanto para cada usuario como dependiendo de la capacidad de comunicación de la interfaz y es lo que se denomina adaptación funcional. La adaptación funcional es un cambio en el nivel de comportamiento para ayudar a la adaptación de la interfaz. Esto incluye la desactivación de determinados elementos y movimiento de éstos en la escala temporal o lógica.

En el nivel de presentación, la arquitectura Oni es suficientemente laxa como para que la adaptación se pueda realizar de forma ortogonal a los mecanismos que introduce. En el nivel funcional, la adaptación queda completamente fuera del ámbito de Oni.

La adaptación funcional, sin embargo, queda completamente fuera de los objetivos y de las funciones de Oni y ésta simplemente propone los mecanismos necesarios para su implementación (mecanismos de capturas de jerarquías y de creación y borrado de componentes).

3.5.8. Protium

Protium [158] es una arquitectura de aplicaciones similar a la de Oni. En Protium las aplicaciones están divididas en dos partes. Por un lado la aplicación que mantiene el estado a largo plazo y se encarga de realizar la lógica de la aplicación. Por otro el visor de la aplicación se encarga de la presentación y tiene el estado necesario para asegurar una respuesta rápida al usuario.

Ambos extremos se comunican mediante un protocolo específico para cada aplicación y el sistema se encarga de hacer la migración y el mantenimiento trans-

parente de vistas. Esta idea surgió con el Blit [97] se perfeccionó con Sam [99] y se desarrolló con Protium .

Aunque el protocolo que hablan visor y aplicación es propio de cada aplicación, está anotado de una forma especial para hacer que no sea complicado escribir una infraestructura genérica y que funcione para nuevas aplicaciones. Además esta anotación permite escribir multiplexores y demultiplexores que se encarguen de usar un sólo canal de comunicación para diferentes visores y servicios, como se puede ver en la figura 3.6.

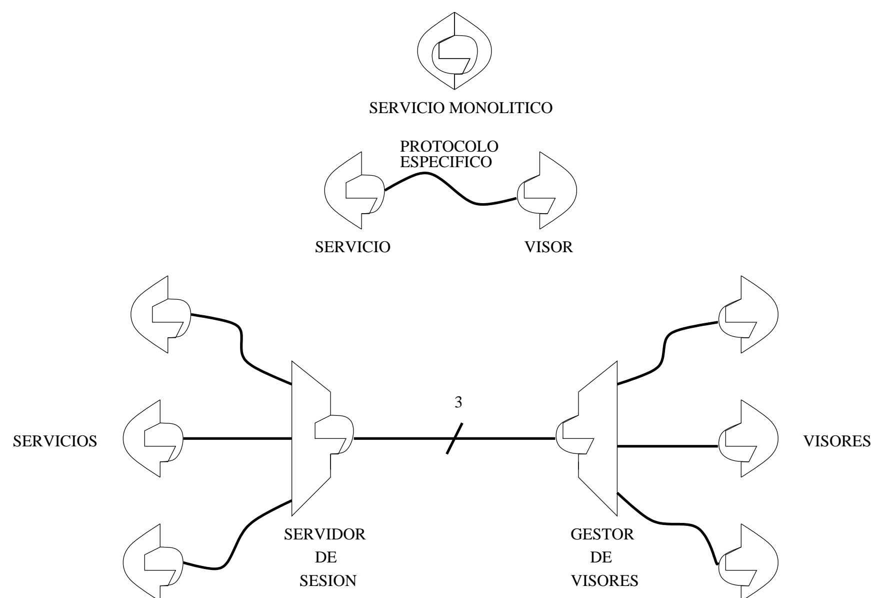

Figura 3.6: *Arquitectura de Protium*

Protium tiene dos desventajas importantes comparado con Oni. La primera es que el trabajo de portar Oni a cada sistema diferente en el que lo queremos usar se multiplica por el número de aplicaciones que deseemos en ese sistema. La segunda desventaja es que el uso de un protocolo propio para cada aplicación hace imposible inspeccionar la interfaz de la aplicación para automatizar tareas de forma genérica e independiente de la aplicación.

3.5.9. Ajax

Ajax [50], "Asynchronous JavaScript and XML", es un framework de desarrollo de aplicaciones web interactivas. Se basa en que la interfaz de usuario es un programa en javascript que ejecuta en el navegador del cliente y que mantiene una comunicación asíncrona con el servidor.

Esto a su vez está construido mediante un mecanismo que permite la comunicación con el navegador de forma asíncrona y sin recargar la página y otro mecanismo que permite recargas parciales de la página web, como se puede ver en la figura 3.7.

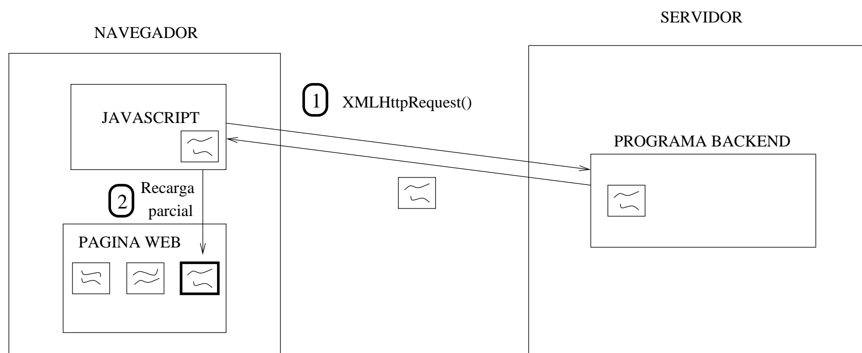

Figura 3.7: *Ejemplo de uso de Ajax*

El resultado es similar al de Protium descrito anteriormente. El navegador del cliente contiene el estado necesario para hacer que la aplicación interactiva tenga un tiempo de respuesta rápido. De forma similar a Protium el problema es la falta de un protocolo común y la dificultad para inspeccionar la interfaz dado que su estado se encuentra dentro del navegador en forma de objetos javascript. Un efecto secundario interesante de esta aproximación, sin embargo, es que obliga a describir lo que se está mostrando en cada momento de forma textual. Esto permite procesar lo que se está mostrando al usuario en ese momento de forma automática. Un ejemplo interesante de uso de esto es el procesado de comandos de voz de Opera, que utiliza una gramática descrita en XML basada en el estándar XHTML+Voice que se encuentra en [8]. Esta gramática procesa

comandos en lenguaje natural y encaja verbos y sujetos con nombres de botones, entradas de texto y enlaces. De esta forma permite la navegación a través de interfaces definidos en Ajax mediante el uso del lenguaje natural.

3.5.10. Scgui

Scgui es similar a Ajax [50] pero sin la programabilidad de javascript. En Scgui la interfaz de usuario en el navegador intercambia descripciones XML de forma asíncrona de la interfaz y de eventos. Es un paso intermedio entre simplemente una descripción de la interfaz en XML y Ajax. Scgui es similar a Oni, pero con dos problemas fundamentales. El primero, es que al no utilizar sistemas de ficheros, carece de nuevo de modelos de protección. Lo mismo sucede con el control y acceso al servidor, que utiliza comandos e interfaces diseñados especialmente para Scgui. El segundo problema es que muchas de las herramientas deben ser reescritas particularmente para esta aplicación. Hay que utilizar una librería especial de Python para listar los contenidos de una interfaz de usuario. Esto es gratis en Oni, pudiéndose hacer con el explorador de ficheros, o con `du -a` o `find`.

3.5.11. XML11

XML11 [113] es un protocolo inspirado en el de X Window System [121]. El navegador web funciona de forma similar a un servidor de X Window System. La aplicación cliente que ejecuta en el servidor web intercambia descripciones en XML con un motor en javascript que se encarga de dibujar la interfaz y manejar la interacción con el interfaz en el navegador web generando descripciones dinámicas del mismo en HTML mediante Ajax [50]. Las aplicaciones pueden estar escritas en java utilizando Swing [2] o AWT [1].

Una de las aplicaciones AWT que existen se llama WeirdX y es un servidor de X Window System escrito completamente en Java y que se puede compilar mediante esta infraestructura para utilizar XML11. De esta forma una aplicación

ejecutando en el servidor web u otra máquina en el lado del servidor puede hablar el protocolo X con WeirdX que a su vez hablará XML11 con el motor en javascript ejecutando en el navegador web. Un ejemplo de esto se puede ver en la figura 3.8.

Uno de los usos de XML11 es dar soporte a aplicaciones escritas en Java que pueden utilizar un compilador cruzado que genera código en javascript basado en Ajax.

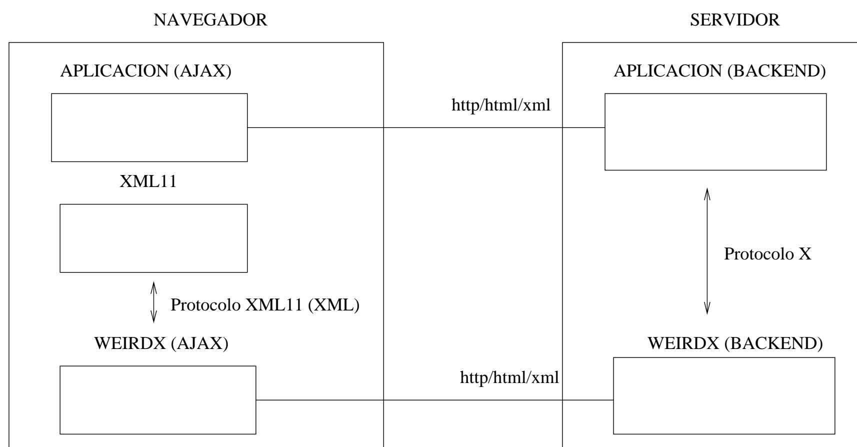

Figura 3.8: *Ejemplo de aplicación usando XML11*

XML11 cuenta con la ventaja de que la interfaz que genera es de texto y por tanto puede ser reificado. Además la existencia de navegadores web con soporte de javascript en casi todos los sistemas operativos hace que las aplicaciones escritas en XML11 sean muy portables.

Sin embargo, todos los demás problemas existentes en X Window System se reproducen, pues XML11 replica el diseño del protocolo X. Además a esto se añade la complejidad de nuevos elementos que añaden problemas a la hora de depurar programas y establecer su seguridad.

3.6. Protección

En este apartado se presentan los sistemas que tienen algún mecanismo de protección de propósito general que permita compartir partes del interfaz. Salvo Mux, que en realidad se encuentra en esta sección por razones históricas (es el precursor de otros), todos los sistemas de este apartado utilizan de sistemas de ficheros. Esto no es casual. Los sistemas de ficheros permiten exportar datos e interfaces de control a la red de forma segura y particionable. Al ser una metáfora sencilla y fácil de entender, es más sencillo conseguir un sistema seguro y poco obstrusivo, lo que influye positivamente en la seguridad [123].

3.6.1. Mux

Mux viene de MPX. MPX era un software experimental para un terminal gráfico del que sólo hubo una versión que construyó Dave Ditzel. El hardware de este terminal era único y fallaba. Después se hizo una reescritura completa para el terminal Blit. MPX actuaba simplemente como multiplexor para terminales, conocidos como “glass ttys”. Mux fue un desarrollo posterior que se convirtió en el programa de control de la máquina. Los terminales se convirtieron en ventanas editables.

Mux se basaba en la idea de que los sistemas de ventanas deben ser transparentes [100] para el usuario, es decir, sólo deben encargarse de multiplexar el sistema operativo en ventanas de la forma que menos distraiga al usuario y sin encargarse de dar mecanismos a éste salvo los mínimos necesarios para realizar esta tarea. Como consecuencia de esta filosofía, Mux tenía una interfaz minimalista basada en menús que permitía borrar, redimensionar mover y ocultar ventanas. Mux fue el primer sistema de ventanas para Unix programado en Bell Labs y fue reemplazado por $8\frac{1}{2}$.

3.6.2. $8\frac{1}{2}$

$8\frac{1}{2}$ [101] (originalmente recibió el nombre de 8.5 hasta la creación de unicode) se basaba en algunas ideas de Mux. Su principal innovación era la de exponer el sistema de ventanas como un sistema de ficheros, aprovechando las ideas de Plan 9 que permiten tener espacios de nombres separados para cada proceso. Así, $8\frac{1}{2}$ multiplexa los ficheros de acceso los recursos, como pueden ser el ratón, la consola y la pantalla mediante el uso de ficheros.

Al programa cliente se le presentan diferentes ficheros en diferentes ventanas que crea el sistema de ventanas. Estos ficheros le dan acceso a los recursos de forma transparente. Por ejemplo, uno de estos ficheros es */dev/mouse*. El programa cliente lo leerá para obtener eventos del ratón. Si el programa ejecuta directamente sobre Plan 9 el fichero se lo suministrará el kernel y los eventos se obtendrán siempre que el ratón se mueva. Sin embargo, si el programa ejecuta en una ventana de $8\frac{1}{2}$, los eventos de ratón sólo le llegarán a la ventana que tenga el foco. De esta forma, el ratón se comparte entre las diferentes ventanas de forma transparente. Algo similar ocurre con la consola o la pantalla.

3.6.3. Rio

Rio [79], el sistema de ventanas de Plan 9 implementado por Rob Pike es descendiente de $8\frac{1}{2}$ [101]. A diferencia que su antecesor, $8\frac{1}{2}$, *rio* fue escrito en Aleph, un lenguaje concurrente desarrollado en Bell Labs. Como consecuencia de la disminución de personal, sin embargo, el mantenimiento de las librerías de Aleph se hizo demasiado arduo y éste desapareció. Rio fue reescrito en C, utilizando la librería de hebras de Plan 9, descendiente directo de las capacidades concurrentes de Aleph.

Igual que $8\frac{1}{2}$, rio multiplexa las capacidades gráficas, textuales y eventos de ratón y de teclado mediante ficheros. Un directorio representa una ventana y en él se encuentran ficheros que representan el contenido textual, gráfico, los eventos de ratón y de teclado asociados a esa ventana.

La principal diferencia entre Rio y $8\frac{1}{2}$ es que $8\frac{1}{2}$ analizaba los comandos de dibujo (textuales) y los reescribía. Sin embargo, en Rio, los píxeles se dibujan directamente en memoria, con una intromisión mínima del sistema de ventanas. Es una diferencia menor realizada por razones de eficiencia y que no representa un cambio significativo para nuestros argumentos. Otro cambio es que en el entorno de rio, el modelo gráfico cambió del sistema de bitblt [102] a un modelo Porter-Duff [108] con soporte completo de color.

Estas mismas ideas reescritas en una variante especial de Scheme dieron lugar a Montage [64].

Aunque en líneas general es cierto que el sistema de ventanas debe hacer lo mínimo posible para multiplexar la interfaz de usuario que representa, la línea que trazan estos sistemas de ventanas creemos que se encuentra a demasiado bajo nivel. Los eventos gráficos que deben cruzar la red (dibujos de rectángulos basados en un álgebra de Porter-Duff) requieren un ancho de banda innecesariamente alto. Además, al ser de un nivel de detalle tan bajo, su reinterpretación con el fin de adaptarlos a sistemas que no tienen los recursos gráficos o de entrada/salida tiene un coste demasiado alto o simplemente no es posible.

Oni utiliza una aproximación similar a rio, pero los elementos que exporta a través del sistema de ficheros son de más alto nivel evitando todos estos inconvenientes.

3.6.4. Sam

Sam [99] es un sistema de ventanas y editor. El sistema de ventanas de Sam es similar a Rio, con una interacción con el ratón y menús muy similar. Sam es similar a *ed* o *vi*, con un lenguaje de comandos, expresiones regulares y direcciones. Es un antepasado de Protium [158] y está dividido en dos partes, la interfaz y la lógica de aplicación. Ambos se comunican mediante un protocolo basado en cadenas de texto.

3.6.5. Acme

Acme [96] es un sistema de ventanas con paneles de texto que realiza la función también de edición y de interfaz de comandos del sistema. Acme sirve también un sistema de ficheros que permite la interacción con el editor y permite también utilizar un lenguaje de edición automático similar al de ed [74], vi [80] o sam [99]. El interfaz de Acme es de tipo typescript, es decir, cualquier texto puede ser utilizado como comando, algo a mitad de camino entre una interfaz de linea de comandos y un editor.

Respecto de sam o Protium, acme es un paso adelante en cuanto a que tiene un sistema de ficheros que permite exportar partes de la interfaz con un modelo de protección sencillo, como Oni. Por otro lado, acme es sólo textual y no proporciona ningún método para el soporte transparente de vistas. Sin embargo, utilizando los mecanismos generales de Plan B, se podría escribir un elemento arquitectónico que mantuviese sincronizadas las vistas para acme similar al de Oni. De nuevo, el problema principal es que acme no soporta imágenes.

3.7. Mantenimiento transparente de vistas y migración

En esta sección se han incluido también sistemas que incluyen algún tipo de transparencia de ubicación, aunque no puedan soportar por sí mismos varias vistas ni migración de la interfaz de usuario.

Hay una familia importante de sistemas, como pueden ser DPS, NextStep, NeWS y Quartz, que utilizan una separación entre lógica y presentación basada en un lenguaje de descripción de la presentación, como PostScript o PDF. Todos estos sistemas utilizan toolkits o frameworks con los problemas ya comentados. Sin embargo, alguno como Quartz si que tienen un diseño arquitectónico en el que estos frameworks encajan. A pesar de ello, no llegan a establecer servicios suficientemente independientes como para poder separar las responsabilidades

de una forma clara que permita componerlos y reificar los interfaces gráficos y que posea en general las características que le pedíamos al sistema.

La idea de utilizar un sistema gráfico como PostScript o PDF es buena, porque aumenta el nivel de abstracción y permite independizar el gráfico del sistema en el que renderizar. Sin embargo, ambos son sólo un sistemas de imágenes y sigue estando a un nivel de abstracción demasiado bajo para nuestros propósitos. Además, establecer una arquitectura que encaje el nivel de programación, los servicios y esta arquitectura gráfica, no es sencillo, como demuestran las diferentes aproximaciones que ha habido.

Sólo Quartz ha conseguido separar completamente el sistema de ventanas del modelo gráfico, y aun así, el Quartz Compositor [18] no ofrece mecanismos que permitan transparencia de ubicación, ni permite interoperabilidad, a no definir un protocolo o interfaz independiente al que se pueda acceder a través de la red.

Quartz es simplemente un sistema de rendering en lugar de un sistema de ventanas completo.

Mención aparte merece NeWS, que llevó algo más lejos estas ideas hasta llegar a tener una cierta reificación de la interfaz en sus objetos PostScript, aunque de nuevo, esta reificación se obtiene a través de una interfaz de programación nueva. Esto significa que es necesario escribir código contra el API que define NeWS y enlazarlo contra este interfaz de programación para obtener y manipular el estado de los objetos. No hay un protocolo genérico similar al que defino Oni o X Window System [121].

3.7.1. DPS o Display PostScript

En 1987 NeXT Computer Inc. desarrolló, basándose en prototipos preliminares de Adobe DPS un sistema de dibujado en la pantalla basado en PostScript [88]. Este sistema extendió PostScript, un lenguaje originalmente diseñado para mantener diferentes contextos gráficos, con el fin de incluir capacidades interactivas y de programación, mediante la capacidad de enlazar código PostScript en el interior de un programa en C mediante un mecanismo llamado

pswrap.

La versión original de DPS no incluía un sistema de ventanas, sino que utilizaba una implementación de X Window System [121]. Este sistema resultaba extremadamente complicado por la existencia de dos APIs, el de X Window System y el de DPS. DPS actuaba como un sistema de bajo nivel de dibujado que X Window utilizaba para dibujar los objetos en pantalla. Este sistema resultaba demasiado complicado de usar y NeXT decidió incluir ventanas en DPS, lo que resultó en NeXTStep Windowing System.

NeXTStep utilizaba pswraps en C para encapsular elementos gráficos, a su vez encapsulados en objetos ofrecidos en una librería. Este sistema al final no es más que un toolkit que utiliza PostScript como lenguaje de dibujado de alto nivel. El uso de PostScript para el dibujado ofrece ciertas ventajas, como la posibilidad de escalar los gráficos, pero el uso de un toolkit tiene todos los problemas ya comentados.

3.7.2. NeXTStep

NeXTStep, es un sistema operativo multitarea y orientado a objetos [142] que construyó NeXT Computer, Inc [15] [126]. El interfaz de usuario es un sistema de ventanas basado en DPS, aunque reescrito y extendido para aprovechar la orientación a objetos de NeXT. Se han añadido algunos comandos para extender DPS, de forma similar a NeWS [61], aunque más simple. La librería del sistema de ventanas utiliza PostScript [88] para dibujar los elementos gráficos. Estos elementos están envueltos en *pswraps* y se presentan al programador como objetos.

3.7.3. Quartz

Quartz [17] [28] es el nombre que recibe la arquitectura de gráficos de Mac OS X [47]. Quartz es un heredero de NeXTStep que en lugar de utilizar PostScript utiliza PDF [32], el resultado de interpretar PostScript. En este sistema la

posibilidad de combinar código en la parte de PDF es menor que en DPS pero el modelo arquitectónico es similar. La principal diferencia es que se utiliza la separación entre la presentación de la aplicación y su render en pantalla para introducir efectos de aceleración de la tarjeta gráfica.

Quartz está basado en dos partes, el Quartz Compositor, que es un sistema de ventanas [18] que se ocupa de componer los bitmaps que genera cada aplicación. Las aplicaciones generan sus bitmaps utilizando diferentes modelos de imágenes, OpenGL, Quartz 2D o QuickDraw y el Quartz Compositor se ocupa de componer estas imágenes utilizando la aceleración de la tarjeta gráfica, lo que permite realizar efectos gráficos 3D y 2D con las ventanas de forma independiente a la aplicación.

Por otro lado, Quartz 2D es la librería asociada, que se encarga de dibujar texto e imágenes. El modelo interior que utiliza es muy similar al de PDF, lo que hace fácil generar PDFs y reescalar los gráficos de forma similar a como se hacía en DPS.

Las ideas de Quartz han tenido mucho éxito y algo similar se ha implementado para Linux en Beryl [19] y en Windows Vista en Windows Presentation Fundation [41].

3.7.4. NeWS

NeWS [61], "Network extensible Window System" o sistema de ventanas extensible a través de la red fue desarrollado por James Gosling con un equipo en Sun a finales de los ochenta.

Las ideas de NeWS son similares a las de DPS . NeWS incluye un lenguaje de programación propio con herencia que extiende a PostScript. También incluye una forma de acceder al árbol de vistas de elementos gráficos y mecanismos de propagación de eventos e interrupciones a lo largo de este árbol. Este árbol está basado en lienzos o *canvases* que se pueden componer de forma recursiva con otros elementos gráficos. El código de NeWS ejecuta en un intérprete extendido de PostScript.

Las capacidades de NeWS son prácticamente las mismas que las de DPS.

En [53] se describe una implementación de un cliente para MOVIE, Multitasking Object-oriented Visual Interactive Environment, programado en Caltech sobre una estación Sun. Esta versión se escribió al inicio de NeWS y refleja muchas de las incertidumbres y compromisos que hubo en su diseño.

También se desarrollaron varios Toolkits para NeWS, el más importante de los cuales fue TNT, *The NeWS Toolkit*.

Existen varias versiones de NeWS. Hay una versión de SGI que reemplazó su sistema de ventanas que se llamaba FrameMaker de Openlook . Hay también otras versiones académicas, como la que extendió NeWS para multiprocesador PIX, "Parallel Interactive Executive" [81].

Debido a la popularidad de X Window System [121], que es gratuito, NeWS habla el protocolo X, traduciendo las llamadas a Postscript. Esto generó problemas de velocidad y además algunos programas dependen de las posiciones exactas de píxeles tal y como los describían en el protocolo X.

Como consecuencia, Sun reescribió NeWS hibridándolo con un servidor de X Window System, lo que recibió el nombre de Xnews que resultó en un mal servidor de X Window System con problemas de interpretación de PostScript. Después de esto, poco a poco NeWS desapareció. Las últimas versiones desaparecieron cuando Adobe adquirió FrameMaker y Openlook perdió popularidad frente a Motif, pasando a extinguirse.

3.7.5. X Window System

X Window System [121] [57] lo desarrollaron en colaboración Jim Gettys del proyecto Athena [72] (un proyecto conjunto del MIT, DEC e IBM) y Bob Scheifler del LCS (Laboratory for Computer Science), ambos del Massachusetts Institute of Technology (MIT) en 1984. En 1988 una entidad llamada "X Consortium" tomó el testigo del desarrollo y la estandarización, tarea que realizó hasta 1996, pasando su tarea a "The Open Group" que finalmente formó la fundación X.org que controla X window hasta hoy.

La especificación de la X Window System es pública, lo que ha permitido la existencia de muchas implementaciones tanto de los clientes como de los servidores. El nombre de X deriva de su predecesor, un sistema de ventanas llamado W, la letra que antecede a X en el alfabeto. W ejecutaba sobre el sistema operativo V y funcionaba de forma similar a X, aunque su protocolo de comunicación de clientes y servidores era síncrono.

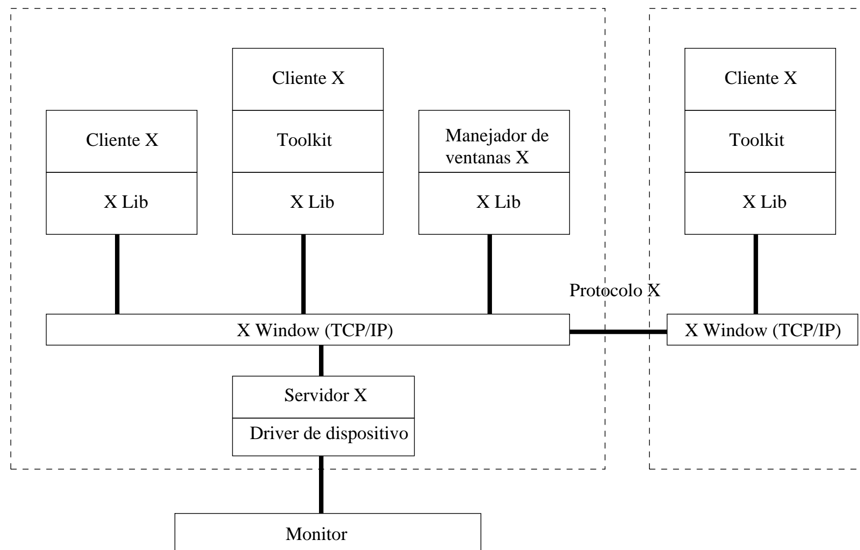

Figura 3.9: *Arquitectura de X Window System*

La arquitectura de X Window System, (ver figura 3.9) se compone de tres elementos fundamentales, un protocolo basado en TCP/IP, X, clientes X y servidores X. Los servidores X se encargan de multiplexar y abstraer el hardware de la máquina del usuario. Los clientes, son las aplicaciones se comunican con el servidor mediante el uso del protocolo de X para realizar operaciones gráficas de bajo nivel y manejar algunos elementos de alto, como el portapapeles o las ventanas. Para programar las aplicaciones se utilizan primitivas de X Lib [57], una librería que abstrae el protocolo para comunicar clientes y servidores.

X Window separa completamente las funcionalidades y responsabilidades de las que se ocupa cada entidad. Por ejemplo, no se marca ninguna política para la

colocación de las ventanas ni su infraestructura, como bordes, redimensionado, etc. Con ese fin existe un programa llamado *manejador de ventanas* (X Window System manager) separado del servidor aunque sea un cliente de X con ciertos privilegios.

Como X Lib es una librería de muy bajo nivel, resulta complicado, tedioso y difícil, programar directamente sobre ella. Por esta razón surgieron muchas librerías de widgets para manejar entidades gráficas de más alto nivel. De estas librerías hemos hablado ya en la sección 3.3.

La idea de utilizar un protocolo común para comunicarse con la interfaz de usuario es fundamental. A pesar de haberse añadido extensiones al protocolo para tratar de agilizar las operaciones gráficas, ver por ejemplo [95], esto no ha resuelto el problema del uso a través de la red que está limitado por el tiempo de ronda [71]. En gran parte de las redes modernas, la mayor limitación de la red es el tiempo que tarda el paquete al pasar a través de la red. Esto significa que si se realizan muchas operaciones en las que hay que esperar a la respuesta para continuar, como puede ser una RPC síncrona, el sistema se hace inusable. El verdadero problema de X Window System es que el nivel de abstracción del protocolo es demasiado bajo.

Como consecuencia el número de operaciones (normalmente operaciones gráficas de bajo nivel) es muy alto, lo que hace inutilizable este protocolo en redes con mucho retardo.

También consume mucho ancho de banda, lo que es malo también para las redes con poco ancho de banda. Esto empeora si queremos añadir mecanismos de reubicación o múltiples vistas utilizando directamente el protocolo.

Además de esto, los mecanismos de seguridad y autenticación de X window han sido y son particularmente vulnerables. Son famosos los casos de fallos en estos mecanismos que han abierto brechas de seguridad importantes [49]. Una descripción de todos estos fallos se puede ver en [71].

3.7.6. Fresco

Fresco [109] (antes llamado Berlín) es un intento de reemplazar el X Window System [121] mediante un sistema capaz de aprovechar mejor las capacidades gráficas de los terminales y mediante la utilización de un protocolo que exporte elementos gráficos de alto nivel. Sus objetivos son similares a los de Oni. Sin embargo, eligieron CORBA como bus de comunicaciones. Como consecuencia de esto, el modelo de seguridad tenía no esta claro [21] y el sistema se complicó hasta hacerse inmanejable para los desarrolladores. Se paró de desarrollar, aunque se hizo una propuesta para sustituir CORBA por Caprice, una arquitectura que intentaba proveer el subconjunto de mecanismos que se utilizaban de CORBA en Fresco. Caprice, nunca llegó a terminarse y Fresco se abandonó.

Oni es similar a Fresco, pero utilizando un sistema de ficheros para exportar lo que en Fresco se llama "scene graph" (grafo de la escena). La utilización de un sistema de ficheros resuelve todos los problemas de Fresco. El sistema de nombrado de CORBA se sustituye por el sistema de nombrado inherente al sistema de ficheros, los mecanismos de seguridad, tanto autenticación como control de acceso esta claro ahora. Además, se pueden usar herramientas ya existentes, como `cat` para obtener el estado de la interfaz.

3.7.7. Photon microgui

Photon [115] es el sistema de ventanas de Qnx, [114] un sistema operativo basado en un microkernel escrito por Gordon Bell y Dan Dodge en la universidad de Waterloo en 1980. Qnx está basado en paso de mensajes entre diversos programas, llamados servidores. El microkernel ofrece los servicios básicos necesarios para este paso de mensajes.

En Photon la interfaz de usuario se modela como una serie de capas rectangulares puestas unas delante de otras a diferentes niveles Los niveles representan profundidad, considerándose "delante" lo que se encuentra más cerca del usuario y "detrás", más lejos. La idea es similar a un sistema de ventanas con ventanas

solapables. En realidad es más general, al permitir romper la interfaz de una aplicación en diferentes rectángulos.

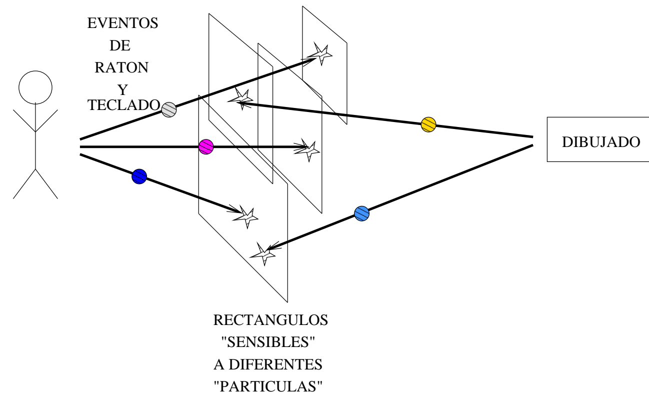

Figura 3.10: *Sistema de capas de Photon*

Estos rectángulos rectángulos son sensibles a diferentes tipos de eventos. Los eventos gráficos, como puede ser el dibujado de una línea, viajan a través los rectángulos de atrás hacia adelante, mientras que los eventos de interacción, como el ratón y el teclado viajan desde delante hacia atrás. Dicho de otra forma, los eventos gráficos son como los fotones en un retroproyector, atrapados (y representados o no) por rectángulos que se interpusiesen entre la fuente y el usuario. En cambio, los eventos de interacción podrían ser como los fotones de un puntero láser en manos del usuario, delante del retroproyector. Un esquema de esta idea se puede ver en la figura 3.10

Esta arquitectura basada en rectángulos solapables, permite la generación y recogida de eventos gráficos y de interacción de forma descentralizada y flexible. Los eventos se propagan como mensajes, que en Qnx es la forma estándar de comunicación entre procesos.

Photon se encuentra en un nivel de abstracción similar a Rio, aunque permite una mayor descentralización por su esquema de paso de mensajes. Al encontrarse a este nivel de abstracción, los problemas de Rio se repiten. A través de la red viajan eventos de demasiado bajo nivel, lo que hace más difícil la adaptación

y reinterpretación de éstos, además de no ser demasiado eficiente en el uso de ancho de banda.

3.7.8. VNC

VNC [143] significa Virtual Network Computing. VNC se desarrolló originalmente en Reino Unido en los laboratorios de AT&T de Cambridge. VNC está basado en el concepto de framebuffer remoto, es decir en el uso de un protocolo que permite al servidor actualizar un framebuffer que está en el cliente.

Un servidor exporta el puntero de ratón, la entrada de teclado, el portapapeles y actualiza la imagen del escritorio a través de la red. Un programa cliente permite la utilización de este escritorio.

VNC hace uso de varios métodos de compresión como ya se mencionó en 2.1 y tiene el problema de ser lento en redes con gran latencia o poco ancho de banda. Además, los métodos de compresión son lentos en sistemas con poca capacidad computacional, como puede ser un teléfono móvil. por último, VNC no permite descomponer las interfaces de usuario, ya que no tiene noción de la estructura de la interfaz de usuario que exporta.

Capítulo 4

Descripción de la arquitectura

En este capítulo describimos cómo se traducen los requisitos de diseño que se describieron en el capítulo 2 en el diseño de la arquitectura propuesta Oni.

4.1. Requisitos de diseño

Los requisitos de diseño específicos de nuestra arquitectura presentados anteriormente son:

- Interoperabilidad entre dispositivos heterogéneos y adaptación a diferentes capacidades.
- Reflexión y programabilidad mediante la reificación, lo que permite adaptarse a la dinamicidad del entorno.
- Adaptación modal, separación de lógica y presentación para adaptar la interfaz a las necesidades del usuario y del entorno.
- Un modelo de protección adecuado a dicho entorno.
- Mantenimiento transparente de vistas y migración lo que permite adaptar la interfaz a las necesidades del usuario.

Sin embargo, además de estos requisitos de diseño propios de nuestra arquitectura, hay otros propios de cualquier arquitectura de interfaces de usuario que analizaremos detalladamente. La forma en la que se implementan estos requisitos generales y los compromisos que establecen, será lo que permita desarrollar adecuadamente los requisitos específicos de nuestra arquitectura. Estos requisitos son:

- Representación del estado de los elementos o widgets.
- Representación de las relaciones entre elementos.
- Creación de metaelementos para establecer relaciones o propiedades de agrupación.
- Elección del conjunto mínimo pero completo de elementos necesario.
- Elección de un lenguaje que permita expresar los eventos de interacción del usuario con los widgets.
- Elección del lenguaje de eventos de modificación de la interfaz a los programas clientes y vistas.
- Interfaz de programación para los programas cliente del sistema de ficheros.

Estos requisitos hay que interpretarlos en el contexto de su implementación en un sistema de ficheros de widgets. Estos widgets o elementos tendrán su representación en el sistema de ficheros lo que se reflejará en la librería de programación que utilizarán los clientes.

En este capítulo veremos cómo se codifican las diferentes propiedades de los widgets y las relaciones que existen entre ellos. En él analizamos cada uno de los requisitos específicos de la lista anterior y veremos las diferentes alternativas de diseño en el contexto mas amplio de los requisitos propios de nuestra arquitectura. Estas alternativas de reflejarán de forma diferente en la estructura y semántica del sistema de ficheros.

4.2. Planteamiento básico

El propósito de la arquitectura, descrito el capítulo 2, es ofrecer un servicio independiente de interfaces de usuario a las aplicaciones con las propiedades descritas en la sección 4.1 Como consecuencia, cualquier diseño para la arquitectura de Oni tiene que cumplir los requisitos y repartir una serie de responsabilidades propias de una interfaz de usuario con múltiples vistas. Concretamente, estas responsabilidades son:

- Anuncio de nuevas vistas
- Acceso de las vistas a los clientes y viceversa
- Transparencia de localización de vistas y/o clientes

Estas responsabilidades se podrían traducir en varias arquitecturas distintas con diferentes entidades. Desde el primer momento, teníamos en mente el uso de un sistema de ficheros para exportar cada vista, como interfaz entre la interfaz de usuario basada en widgets y los programas cliente. El uso de un sistema de ficheros hace que el sistema sea interoperable y programable con las herramientas que ya existen para interactuar con ficheros. Sin embargo, hay varias formas de hacer esto y simultáneamente mantener la transparencia de localización.

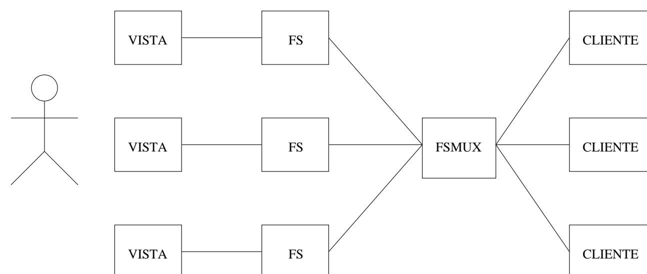

Figura 4.1: *Arquitectura con sistema de ficheros multiplexor común*

1. Se puede utilizar una entidad que actúe como multiplexor, interponiéndose entre el sistema de ficheros y los programas cliente y haciéndoles creer que

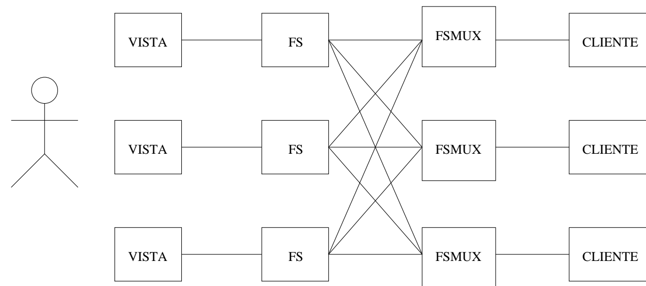

Figura 4.2: *Arquitectura con sistema de ficheros multiplexor por cliente*

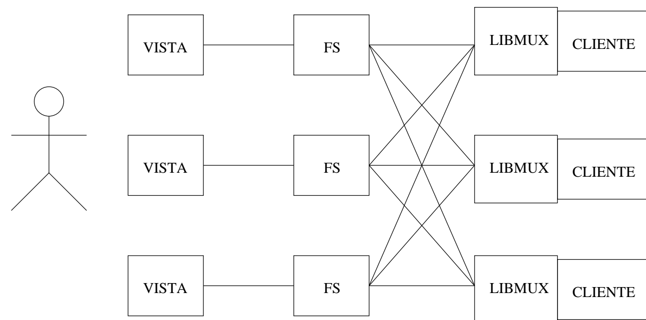

Figura 4.3: *Arquitectura con librería multiplexora*

hay una sola vista. Este multiplexor, se puede construir como un sistema de ficheros a su vez. En las figuras 4.1 y 4.2 se encuentran representados las dos formas posibles de utilizar este multiplexor, ejecutando un multiplexor por cliente o teniendo un multiplexor común para todos.

En ambos casos se tiene el problema de que los sistemas de ficheros multiplexadores precisan ser excesivamente complicados. Esto es debido a que, como sucede con los ficheros de eventos, no se trata simplemente de la sincronización de sistemas de ficheros, sino que hay que realizar tareas más complejas. Además, algunos elementos de alto nivel, como la selección en un panel de texto son locales a cada interfaz, con lo que habría casos especiales de eventos y contenidos que no habría que copiar.

Para no añadir demasiada complejidad al multiplexor, sería mejor utilizar

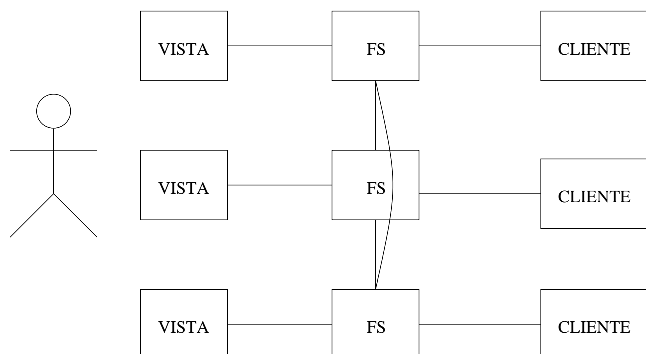

Figura 4.4: *Arquitectura con sistema de ficheros p2p*

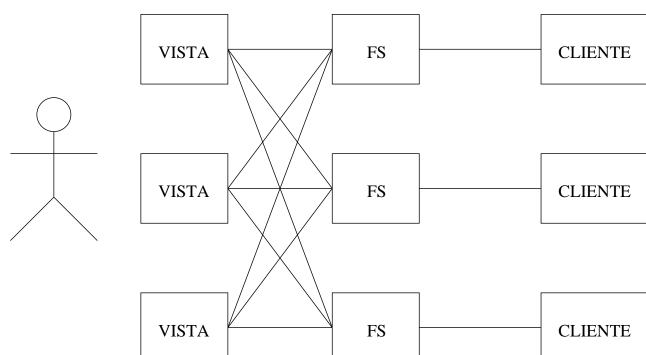

Figura 4.5: *Arquitectura con sistema de ficheros asociado a múltiples vistas*

la arquitectura de la figura 4.2 con un multiplexor por cliente, pero aún así, el hecho de que sea un sistema de ficheros lo complica innecesariamente.

2. El segundo tipo de arquitectura se refleja en las figuras 4.3 y 4.4. En ambos casos los clientes se comunican con un sólo sistema de ficheros y éste se encarga de mantener de forma transparente las vistas.

   En la arquitectura representada por la figura 4.3, las vistas se comunican con el sistema de ficheros. Para esta comunicación haría falta un nuevo protocolo que atraviesa la red, con un modelo de protección, un sistema de anuncio y acceso. Con lo que recreamos el problema que había en el interfaz entre sistema de ficheros y cliente. Simplemente hemos incluido

la funcionalidad del multiplexor en el sistema de ficheros. Esto complica el sistema de ficheros, convirtiéndolo en algo similar al sistema de ficheros multiplexor que ya hemos descartado por ser demasiado complicado.

3. La arquitectura por la que nos hemos decidido finalmente es la de la figura 4.5. El multiplexor es una librería enlazada con el cliente. De esta forma, la multiplexación se da en la interfaz que ya está bien definida a nivel de protocolo, modelo de protección y sistema de anuncios: el sistema de ficheros. Además, al empotrar en la librería de programación la multiplexación, se traslada el control de cómo funciona la replicación allí donde hay información sobre los widgets, en la librería de widgets.

   Aún así y para simplificar la implementación, existe igualmente una cierta separación entre el visor o la vista, es decir, la interfaz de usuario y el sistema de ficheros. Aunque esta interfaz es interna al programa y no se expone al programador de aplicaciones.

   El acceso y anuncio se realiza en el interfaz del sistema de ficheros, lo que permite utilizar mecanismos estándar de Plan B diseñados con este fin.

4.3. Arquitectura propuesta para el sistema

La arquitectura propuesta para el sistema se puede ver en la figura 4.6. Está compuesta de tres elementos. Por un lado, el servicio de interfaces de usuario ofrecido por la arquitectura propuesta, Oni. Una instancia de este servicio ejecutará en cada pantalla. A él se accede a través del segundo elemento, el sistema de ficheros. El sistema de ficheros se encarga de abstraer la interfaz de usuario que exporta el servicio de widgets. En tercer lugar se encuentra el programa o aplicación que utilizaría el sistema de ficheros y que se encarga de implementar la lógica de aplicación.

Es importante observar que el sistema de ficheros es sólo una interfaz de programación. Los ficheros a los que nos referimos, no son ficheros respaldados

por un disco, sino entidades dinámicas que forman parte de una abstracción, de forma similar al sistema de ficheros **procfs** en un Unix [76].

Cada una de las entidades definidas en nuestra arquitectura tiene una serie de responsabilidades establecidas, una relación clara y una interfaz bien definida con el resto.

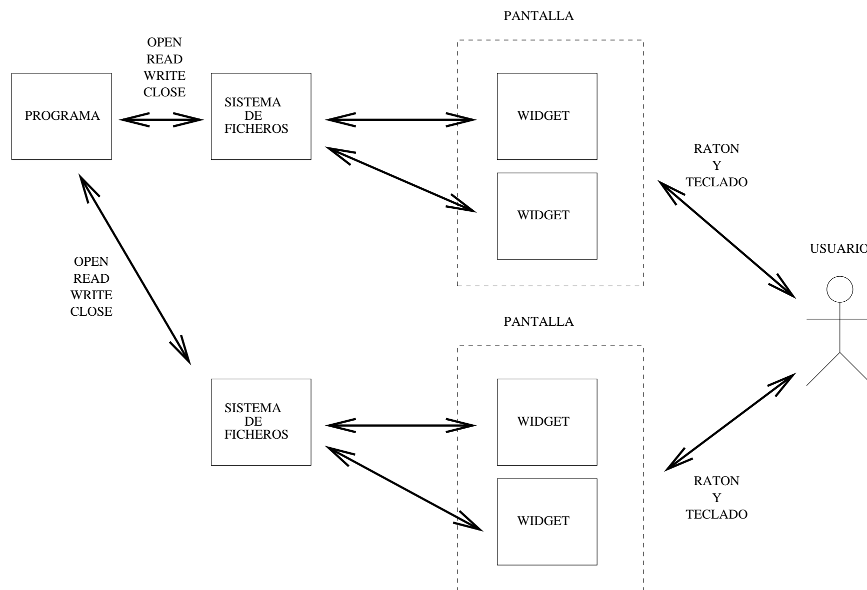

Figura 4.6: *Arquitectura propuesta, Oni*

El servicio de interfaces de usuario tiene como responsabilidad mostrar los objetos gráficos que forman parte de la interfaz e interactuar con el usuario. Tanto los objetos gráficos como los eventos procesados y filtrados se muestran como un sistema de ficheros. Esta interfaz sencilla unida al uso de eventos de alto nivel contribuyen a la portabilidad. Permiten escribir aplicaciones cliente en cualquier lenguaje y en cualquier sistema operativo que posea facilidades para manejar ficheros.

Por otro lado, los eventos que no requieren más que un cambio en la interfaz, como por ejemplo el cambio de lugar de un buffer o ventana, no salen de la interfaz. De esta forma, sólo se replican los eventos que de verdad cambian el estado de la aplicación.

El mantenimiento transparente de vistas se basa en dos mecanismos.

1. El sistema de anuncios de Plan B para anunciar nuevas vistas representadas como árboles de ficheros. Este sistema de anuncios es un mecanismo general de Plan B cuyo propósito es anunciar servicios representados mediante árboles de ficheros.

2. Sincronización de las diferentes vistas. De esta sincronización, basada en eventos que generan las vistas con las actualizaciones, se ocupa la librería de programación con la que se enlazan las aplicaciones cliente. El segundo mecanismo tiene también como función abstraer de las operaciones concretas en los ficheros, ofreciendo una interfaz de programación similar a la de cualquier toolkit de widgets.

   De esta forma se podrían utilizar aplicaciones ya existentes. Estas aplicaciones utilizan toolkits contra los que se enlazan. Es sencillo implementar una librería que provea de esta interfaz a las aplicaciones y así no tener que reescribirlas.

Esta librería realiza, pues, una doble función, abstracción y demultiplexación. Se pueden separar ambas funciones, implementando la demultiplexación mediante un programa externo.

Finalmente, el programa usuario se encargará de la lógica de aplicación, manipulando los widgets y procesando los eventos que obtiene a través del sistema de ficheros cuando sea necesario. Este programa ignora la existencia de varias vistas o la localización de las mismas, que podrá cambiar incluso de forma dinámica. De esta forma se cumplen los objetivos de mantenimiento transparente de vistas y relocalización.

4.3.1. Funcionamiento global del sistema

En la figura 4.7 se encuentra representado el funcionamiento global del sistema. A la izquierda se ve un ejemplo de programa cliente. Este programa utiliza

la librería de widgets para acceder al sistema de ficheros, en el centro, que representa al interfaz de usuario, a la derecha. En cada réplica de la interfaz de usuario, que se pueden ver en el centro, arriba y abajo, hay dos jerarquías, o árboles de ficheros una para datos y otra para eventos. En el dibujo se ha representado la jerarquía completa de eventos con una sola caja llamada EVENTS, para no complicar la representación, pero esta jerarquía es una copia de la que cuelga del raíz excluida ella misma, con la diferencia de que no existe ninguno de los ficheros de la jerarquía de datos y en todos los directorios hay un fichero llamado **events**. El programa cliente accede a esta jerarquía para recibir eventos, leyendo de los ficheros.

Cada directorio dentro de ambas jerarquías representa a un contenedor o a un widget. En la figura se pueden ver tres widgets, dos de imagen; **esoriano** y **nemo** y uno de gráficos vectoriales, **bars**.

Por otro lado, se pueden ver dos tipos de contenedores, representando una ventana, **win** y otro representando una columna **col**.

A la derecha se puede ver una representación esquemática del aspecto que tendría el display de las dos vistas.

A continuación pasamos a justificar la elección de cada uno de estos interfaces y formas de representación con más detalle.

4.4. Representación del estado de los widgets o elementos

Para cada tipo de widget se debe elegir una codificación del estado que permita leer y escribir el estado del mismo en cada momento. Con codificación o representación, nos referimos a cómo presentará el sistema de ficheros el estado de los widgets y la semántica empleada para interpretar dicha presentación.

No se debe confundir este estado con los eventos, ya sean de interacción o de actualización. Por ejemplo, el estado de un panel de texto puede ser una cadena de texto, la posición del cursor y el texto seleccionado. Si alguien escribe algo

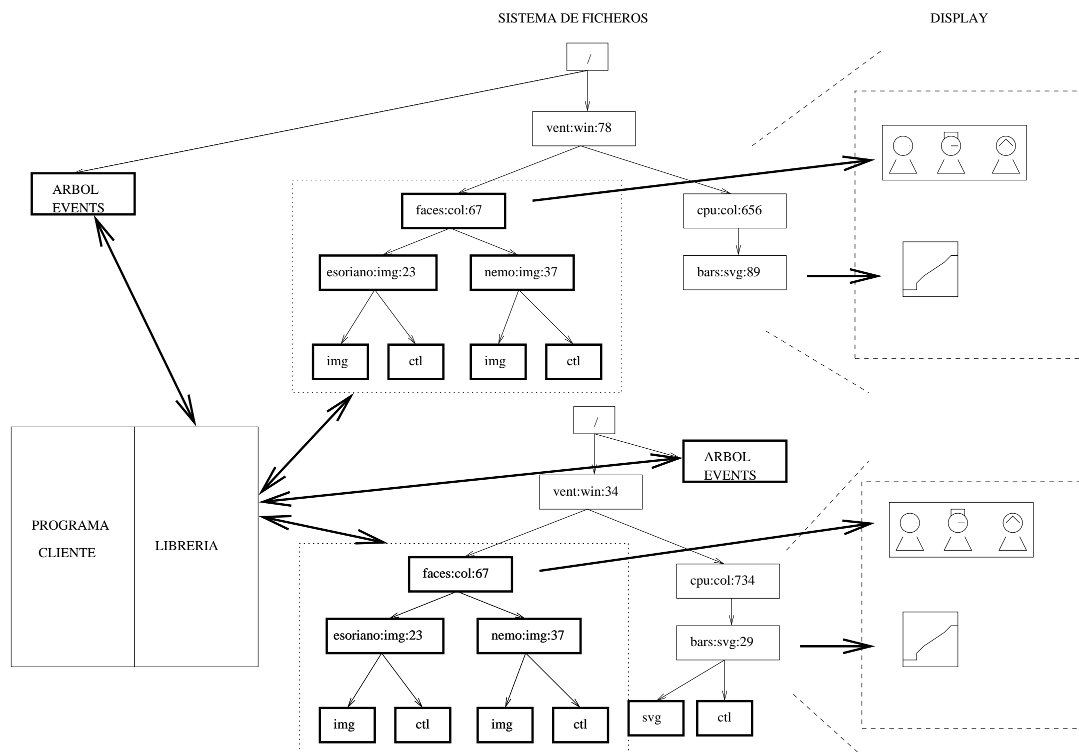

Figura 4.7: *Funcionamiento global del sistema.*

sobre el panel, esa acción modifica el estado aunque además genera un evento. Un evento representa una acción, un cambio. El estado representa el contenido estático de la interfaz de usuario. Para copiar el panel de texto a otro interfaz, se deben realizar las siguientes acciones:

1. Crear otro panel de texto.

2. Copiar el estado del panel a él.

3. Anunciar la existencia del nuevo panel de texto.

4. Generar y manejar eventos de actualización para programas cliente y vistas.

En los pasos 1 y 2, leemos el estado del widget origen y lo escribimos en el widget destino. Estas interacciones las realizamos a través del sistema de

ficheros, luego a través del sistema de ficheros tenemos que ser al menos capaces de:

- Crear widgets.
- Leer el estado de los widgets.
- Escribirlos.

Para poder realizar los pasos 3 y 4 tenemos que ser capaces de distinguir de qué tipos son los widgets e identificar uno concreto para comunicarnos con él. Por tanto, mediante el sistema de ficheros, debemos ser capaces además de:

- Darle un identificador único al widget.
- Darle un tipo al widget.

Darle un identificador único es importante para distinguirlo a nivel programático del resto de paneles de texto que hay en la red, de forma que sea posible más adelante diferenciar de quien vienen los eventos. Este identificador puede ser simplemente un número aleatorio suficientemente grande.

Es importante que el widget tenga un identificador que se pueda leer, para poder interactuar con él a través de una interfaz de voz, y que el usuario pueda identificarlo de forma legible. Un ejemplo de identificador para el panel de texto podría ser "tesis".

El tipo del widget, será simplemente el que identifique qué es el widget. Por ejemplo un panel de texto o un botón.

4.4.1. Ficheros vs. directorios

Hay diferentes formas de representar todos estos elementos en un sistema de ficheros. Por un lado, contamos con el nombre de los directorios o ficheros. Este nombre, utilizado juiciosamente, nos permite representar cadenas de texto. Por otro, contamos con el contenido de ficheros, que nos permite representar de

forma general cualquier estado, ya sean cadenas de texto, objetos binarios como imágenes o, en general, cualquier contenido.

La forma más sencilla de representar un elemento en un sistema de ficheros es utilizar sencillamente un fichero. Esto tiene varios problemas. El primero de ellos es que eso significa que la interfaz de los widgets y de los metalementos es fundamentalmente diferente, ya que unos serían ficheros y los otros directorios ya que los metaelementos sirven para contener a otros en su interior y por tanto forzosamente tienen que ser directorios.

El segundo es que no nos permite separar los diferentes contenidos de forma clara y sencilla. Por ejemplo, el panel de texto tiene por un lado la selección y por otro el contenido en texto del mismo. Si incluimos ambos en un mismo fichero, es más complicado hacer programas que sólo usen la selección, pues antes deberán extraer la selección del fichero, separándola del resto del contenido. Esto nos lleva a que es mejor representar un elemento como un directorio con varios ficheros en el interior que representen el estado. Se podrían usar varios ficheros sin estar incluidos en directorios, pero de nuevo, estableceríamos una diferencia artificial entre los metaelementos y los elementos.

Puesto que estos directorios coexisten y potencialmente puede haber colisiones en los nombres, el identificador único del que hablábamos, debe formar parte del nombre del directorio. Por otro lado, para poder reconocer el elemento de forma sencilla, es también conveniente que el nombre textual forme parte de ese nombre, convenientemente separado del número aleatorio.

Sólo con estos dos elementos, ya estamos seguros de que el nombre del directorio identifica al widget de forma unívoca y legible. Un ejemplo de esto sería “tesis:2345234”. Un interfaz de voz leería sólo “tesis” o “versión uno de la tesis” si hubiese colisión. Las aplicaciones podrían reconocer los números aleatorios y/o el nombre, dependiendo de su tarea.

4.4.2. Representación del tipo

Antes de entrar en el estado, nos queda por representar algo que está a medio camino entre el estado y la identificación, el tipo. Por un lado, no es necesario codificar el tipo del elemento en el nombre, puesto que con lo que teníamos ya podemos reconocer el tipo de forma unívoca, simplemente con un dato representando el tipo en el contenido del widget. Por otro, el tipo es una parte muy importante de lo que realmente es el elemento, hasta el punto de que un elemento creado sin tipo, está en un estado indefinido. Además, dos elementos con diferentes tipos son algo completamente diferente. El panel de texto “tesis” y un botón “tesis” que está asociado a una aplicación que crea una tesis cuando aprietas al botón son cosas fundamentalmente diferentes.

Los metalementos, como veremos más adelante, a diferencia de los elementos, no están completamente definidos por su tipo, sino que el tipo es una propiedad que puede cambiar. Por ejemplo un metaelemento columna puede pasar a ser un metaelemento fila sin que algo fundamental cambie.

A pesar de esto, hemos optado por codificar el tipo del elemento en el nombre, frente a la alternativa que podría haber sido codificar el tipo en un fichero de forma similar al estado. Esto simplifica la implementación, ya que, al crear el directorio para el widget, ya está claro de qué tipo es y no hay estados intermedios indefinidos. Sin embargo, esto nos priva de flexibilidad y complica la posibilidad de cambiar de tipo los metalementos.

El resultado final es que un elemento o metalemento corresponde a un directorio de nombre “Nombre:tipo:ID” donde el nombre es una cadena de texto UTF-8 arbitraria, con las habituales restricciones de los nombres de ficheros, tipo es el tipo del widget, como puede ser “panel” o “button” y ID es un número aleatorio positivo de 32 bits escrito en ASCII. Hemos utilizado como separación el carácter `':'`, pero la elección es arbitraria. Lo hemos elegido simplemente porque no tiene un significado especial para la shell como `'*'` ni para algunos sistemas de ficheros, como podría ser el punto.

4.4.3. Representación del estado

El directorio que representa a un elemento contiene en su interior tantos ficheros como sea necesario para representar el estado de los widgets. Hay dos tipos de ficheros, los que contienen el estado que se escribe literalmente, como puede ser el contenido de un panel de texto. Si queremos que en el interior haya una cadena, se escribe. Si leemos el fichero, obtenemos el estado, tal cual lo hemos escrito.

Otro tipo de ficheros son los ficheros de comandos. En estos ficheros, se escriben comandos que afectan al widget. El resultado de leer el fichero no es necesariamente lo que se ha escrito. Cuando se lee el fichero, se obtienen los comandos necesarios para reproducir el estado del widget. En realidad, el interior del fichero, no contiene datos, sino metadatos o atributos que se pueden cambiar al escribir en el fichero o leer al leer del fichero. Todos los widgets, contendrán un fichero de comandos de nombre **ctl** en el que se podrán escribir comandos que no tengan un fichero particular que les corresponda.

Los ficheros del interior del árbol de widgets se listarán en el orden adecuado para que copiar su contenido a un widget recién creado establezca una réplica. Esto significa que los ficheros de estado irán antes de los de comandos y que los ficheros de estado se listarán en un orden que se corresponda con el orden de representación en pantalla, por ejemplo. Además hay que ser cuidadoso con cómo se expresa el contenido de los ficheros.

4.5. Relaciones entre widgets

La representación en pantalla o en audio de los widgets requiere ordenar los widgets de una determinada manera. Esto se puede expresar como una relación de precedencia. De forma similar, otras relaciones, jerárquicas o de clase se pueden utilizar para expresar otras propiedades de los widgets.

Las relaciones entre widgets pueden ser de varios tipos como se puede ver en la figura 4.8. Hay relaciones de tipo inclusión $A \subset B$, en las que un elemento B

está incluido en otro A. Otro tipo de relación es la pertenencia a un conjunto A $\in$ B, en las que un elemento B pertenece a un conjunto A de elementos. Y luego existen varios tipos de relaciones de agrupamiento añadiendo alguna propiedad espacial, como contigüidad lado a lado.

Estas últimas relaciones podrían establecerse como propiedades de los objetos o de forma independiente dándole entidad a las relaciones. Creemos que es mejor que estas propiedades puedan codificarse en las relaciones, ya que ésto captura mejor su naturaleza. Al fin y al cabo la colocación de los elementos, ya sea en pantalla, ya sea leídos en una lista, es algo temporal que el usuario puede cambiar.

Aunque haya algunos objetos que en una representación concreta puede parecer que resultan más cómodos situados en una determinada posición o con una cierta propiedad espacial, esto no es necesariamente una propiedad multimodal, con lo cual parece más adecuado que esta propiedad se la asignemos a la relación que al objeto, de forma que estas propiedades se puedan manipular de forma independiente y en algunos casos de forma local al interfaz en el que se encuentran.

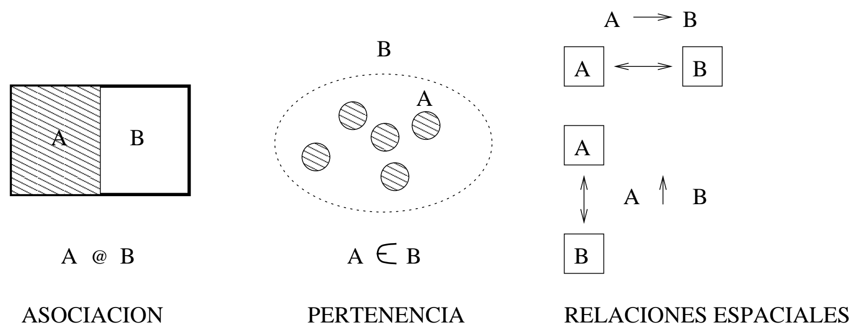

Figura 4.8: *Tipos de relaciones entre elementos*

Tendríamos que expresar tres tipos de relaciones:

- Asociación: A @ B
- Pertenencia: A $\in$ B

- Relaciones espaciales: “al lado”, A $\rightarrow$ B, “encima” A $\uparrow$ B,

El último tipo de relaciones podría especificarse más, en coordenadas 2D o 3D, pero consideramos que esto es responsabilidad de una implementación de interfaz concreto, que conoce el modo y las características de interacción que posee. Oni en su descripción de la arquitectura tiene que establecer estas propiedades de forma laxa, para que una presentación concreta pueda ocuparse de interpretarlas de diferentes formas.

Una vez establecidas las relaciones que queremos codificar, debemos establecer cual es la forma de expresarlas dentro del sistema de ficheros.

4.5.1. Relación de asociación, A @ B

La relación de asociación se produce cuando dos widgets se encuentran uno muy atado al otro, de forma que la interacción en uno sólo produce eventos que se consumen en modificar el otro.

Hemos optado por no codificar de forma explícita la relación de asociación. Si un widget A está asociado a otro B, y no pertenece a un superconjunto al que pertenecen ambos, esto normalmente significa que siempre se presentan asociados y ese widget A debe ser una propiedad del widget B. Un ejemplo de esto son las barras de desplazamiento de los campos de texto. Podría considerarse que una barra de desplazamiento es un widget que está asociada a un campo de texto. Sin embargo, es mejor considerar la barra de desplazamiento una propiedad del campo de texto, de forma que la interacción detallada con el ratón queda confinada en la interfaz gráfica.

En el caso contrario, si se desea disociar completamente la barra de desplazamiento del campo de texto, siempre se puede crear completamente separada y utilizar el mecanismo estándar de generación de eventos y un programa que los reciba y reconozca. De esta forma, existe la posibilidad de manipular ambos widgets de forma independiente, teniéndolos incluso en diferentes máquinas o teniendo sólo uno de ellos replicado. En este caso la relación de asociación no se-

ría más que una implementación particular de un problema que puede resolverse mejor con los mecanismos generales ya existentes.

4.5.2. Pertenencia, A $\in$ B

La relación de pertenencia, sí es importante y se expresa de varias formas diferentes. Por ejemplo, si consideramos la interfaz de usuario de una aplicación como un conjunto de widgets, en seguida vemos que es necesario poder expresar la relación A $\in$ B. Esta relación puede aparecer de diferentes formas, dependiendo del tipo de pertenencia a considerar.

El primer tipo de relación que consideraremos es un tipo de agrupación versátil. Se refiere a una agrupación que define el usuario, la aplicación o ambos y que sirve para definir conjuntos de widgets en general. Un ejemplo de esto puede ser una "sesión". Una sesión sería una agrupación de widgets local a una interfaz que tendría sentido para el usuario.

Por ejemplo, *trabajo* y *ocio*. El usuario podría tener dos grupos de widgets, uno para lo que el usuario define como *trabajo* y uno de *ocio* que dedicase a diferentes tareas. Oni, la arquitectura propuesta, sólo se encarga de apuntar que widgets pertenecen a qué conjunto y de guardar el nombre. La semántica la provee el usuario.

Este tipo de agrupaciones es cómodo para organizar la interacción y sería interesante combinarla con relaciones espaciales. Es local a un display o interfaz, puesto que la controla el usuario, aunque la puede definir la aplicación, y es una entidad independiente que se puede copiar o replicar. Debido a que su comportamiento es similar a la de cualquier otro elemento y que se puede manipular. Una forma sencilla de implementarla es la creación de metaelementos que representen el conjunto. Estos metaelementos realizarán el papel de contenedores. Este papel en el sistema de ficheros lo realizan directorios, que expresan que la idea de contenedor en un sistema de fichero. Les llamamos metaelementos porque abstraen conceptos como el de relación y los reifican en un elemento.

Mediante el uso de metaelementos logramos dos propiedades relacionadas

con la reificación y que son muy importantes, la reificación estructural y la reificación de posicionado. Los metaelementos se pueden copiar, borrar o mover y esta actividad sucederá sobre el metaelemento igual que sobre los elementos del conjunto, como sucede con cualquier directorio. Esta propiedad es la reificación estructural.

La reificación de posicionado, provendrá de añadirles a estos metaelementos información espacial, como se verá más adelante. Por un lado no hará falta situar los componentes, lo que se podrá hacer de forma automática y por otro, existirá una interfaz para manipular este posicionado, cambiando esta información espacial, aunque sea a alto nivel, de forma automática.

Otro tipo de relación de agrupación importante es la de identificación de pertenencia a nivel programático. Este tipo de pertenencia la establece la aplicación y es la forma en que ésta identifica un conjunto de widgets como propios. Es global y debe tener transparencia de red. La aplicación debe ser capaz de identificar un conjunto de widgets después de marcarlos con algún identificador. Si la relación fuese local, solucionar este problema sería tan sencillo como añadir una propiedad del widget que fuese un número aleatorio generado por la aplicación. No obstante, este identificador unívoco no basta, ya que hace falta que la aplicación se dé cuenta de que se ha realizado una copia de una parte de la interfaz, por ejemplo, a un sistema de ficheros que acaba de aparecer.

Para que esto sea posible debe existir la posibilidad de anunciar a los programas existentes en la red la aparición de nuevos elementos y esto se puede realizar de tres formas diferentes, dos de ellas distribuidas y una centralizada.

De forma distribuida, se puede utilizar una dirección de red multicast a la que se envíen los anuncios. Esta estrategia, es la que utiliza, por ejemplo, Jini [25] para anunciar elementos a la red. El uso de un número aleatorio suficientemente grande, para evitar colisiones y anuncios multicast, por ejemplo, resolvería el problema. No obstante, no todas las redes soportan multicast. Si nos encontramos ante redes heterogéneas, empezamos a precisar de proxies que se ocupen de actuar de enlace entre los elementos de estas subredes.

La segunda estrategia que se puede establecer es que el identificador unívoco contenido en el widget que identifica a la aplicación contenga una dirección de red, que puede ser por ejemplo un URI [31] o de forma más sencilla, una dirección TCP/IP [133] y un puerto. En esta aproximación, el servidor de ficheros o un programa externo, se conectaría a la aplicación y se encargaría de avisar a ésta.

Esta aproximación tiene la ventaja de que está soportada en casi todas las redes, pues en casi todas las redes existe alguna forma sencilla de resolver nombres o usar TCP/IP de forma genérica. Tiene también la ventaja de que es muy sencilla de implementar. La desventaja es que el servidor de ficheros, o peor, un programa externo que hay que añadir hace de cliente complicando la relación entre el servidor de ficheros y el cliente. Esto hace más complicada la autenticación y la recolección de basura. Esta estrategia, es la única completamente descentralizada que cumple los requisitos que buscamos.

La tercera estrategia, consiste en utilizar una máquina bien definida como servidor de anuncios. Este servicio de anuncios puede ser un sistema de ficheros, por ejemplo en el que un fichero se bloquee en lectura y se desbloquee con cada anuncio. Esta solución es más sencilla, aunque requiere que haya un servidor de anuncios en la red al que se pueda acceder y que se encuentre bien definido.

Esta última solución es por la que hemos optado por ser más sencilla a pesar de requerir un servidor centralizado. Suponemos que en un futuro próximo habrá acceso a una red con infraestructura en todo momento. Si esto no fuese así, habría que implementar la solución anterior.

La estrategia de anunciar árboles de ficheros asociados a servicios, es una estrategia general de Plan B [27], el sistema operativo en el que hemos desarrollado esta arquitectura, en el que estos árboles llevan un lenguaje de descripción asociado que se puede utilizar para discriminar que servicio se debe usar. En Plan B [27], han existido dos servicios de anuncios, uno multicast y otro que es el que se utiliza actualmente, asociado a un servidor centralizado.

La posibilidad de definir relaciones de pertenencia independientes de la máquina permiten conseguir el requisito de mantenimiento transparente de vistas

y migración. Sin tener una forma de expresar relaciones de pertenencia independientes de la máquina y globales, es imposible lograr este requisito de diseño.

4.5.3. Relaciones espaciales

Hay muchas formas diferentes de codificar relaciones espaciales. Una forma de hacerlo es codificarlas como relaciones entre los elementos y el espacio. Esto equivale a añadir información de posición a los elementos. Sin embargo, esta aproximación tiene dos problemas. El primero es que si cambia la cantidad o forma del espacio disponible la información de posición deja de tener sentido. El segundo es que puede ser necesario reconfigurar la interfaz de forma automática y la información de la que provee solamente la posición no es suficiente. De nuevo, hace falta información de más alto nivel.

Como consecuencia de no tener un espacio estático al que referirnos, la información de posición será intrínseca al elemento y se corresponderá a su relación con otros elementos. Como hemos visto en la sección anterior es necesario añadir metaelementos de agrupación para codificar otras relaciones. Puesto que las relaciones espaciales son volátiles y transitorias y afectan a grupos de elementos, como columnas o filas, es mejor codificarlas en los mismos metaelementos de los que ya hemos hablado, logrando así además una mayor reificación de posicionado.

La solución que hemos adoptado es codificar la posición en el tipo de los metaelementos de agrupación. En cualquier caso, estas relaciones serán sugerencias para la implementación que las interpretará en base a los modos y el espacio disponible.

4.6. Agrupación y codificación de relaciones

Como ya se ha explicado en la sección anterior, Las relaciones de agrupamiento y las relaciones de contigüidad es mejor representarlas por medio de metalementos que actúan como elementos contenedores, para así lograr una

reificación estructural y de posicionado. Esto ya se hace así en la mayor parte de los toolkits, por ejemplo, los *canvas* o lienzos de AWT [1] son metaelementos contenedores de widgets.

Estos metaelementos, los representaremos de forma similar a los elementos normales mediante directorios con la diferencia de que estos contendrán en su interior otros elementos.

Hemos reducido al mínimo posible estos elementos, intentando construir el conjunto mínimo necesario que expresase la relación de agrupación y añadiéndole (siempre se puede ignorar) una relación de contigüidad espacial en dos dimensiones. Este conjunto mínimo, son dos tipos de metaelementos:

- **row**, es decir fila.
- **col**, columna.

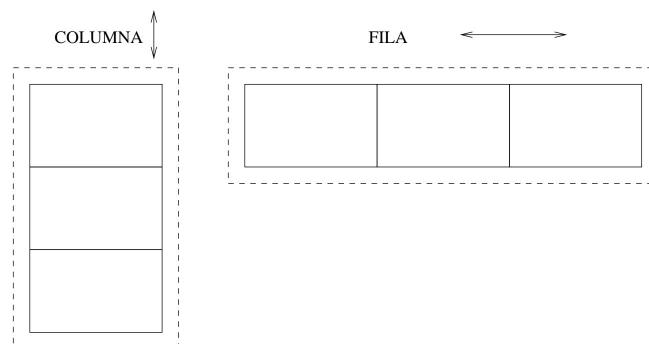

Figura 4.9: *Metaelementos fila y columna; dos relaciones de contigüidad*

Estos dos tipos representan las dos posibles relaciones básicas de contigüidad desde un punto de vista abstracto en un entorno de dos dimensiones, lado a lado, o desplazamiento en una dimensión X y movimiento en la otra dimensión Y. Si se desease, se podría añadir una relación más para tres dimensiones, que podría ser **stack** e incluso otra para cuatro, **timeline**.

Estas relaciones, como ya se ha explicado, no son más que sugerencias para la implementación. Una implementación o instancia concreta, puede que sólo

permita apilar objetos en una dimensión, o el modo empleado puede no tener dimensiones inherentes. Por ejemplo en una interfaz de voz habrá que utilizar estas sugerencias a nivel abstracto, como sugerencias de dos tipos de contigüidad.

4.7. Elección del conjunto mínimo de elementos

A la hora de elegir el conjunto de widgets que existirá en esta arquitectura hay varios compromisos de diseño.

El conjunto de widgets tiene que ser lo suficientemente amplio como para permitir desarrollar cómodamente una aplicación. En un extremo tenemos una aproximación similar a la de Protium [158] en la que la aplicación entera se considera un elemento básico. Al otro tenemos X Window System [71] una interacción de demasiado bajo nivel y casi sin procesado de los eventos.

Oni se encuentra a mitad de camino. Intenta utilizar un protocolo general, como X Window System, pero conteniendo la interacción de bajo nivel en el interior de la interfaz. Además intentamos dividir las aplicaciones en componentes reutilizables para no terminar con la interfaz completa embebida en Oni, como le pasa a Protium. Por ello hemos intentado definir un conjunto mínimo de widgets que expresen la funcionalidad completa necesaria. Si hiciese falta, se podrían añadir nuevos widgets, siempre intentando que fuesen generales, pero añadir un widget supone implementarlo en todas las instancias del interfaz.

Recordemos en cualquier caso que un widget a este nivel arquitectónico representa una funcionalidad, no una instancia concreta de la misma. Por ejemplo, un botón en una interfaz de audio, se traduciría, por ejemplo en un comando de voz. De esta forma se mantiene la separación entre presentación y lógica de aplicación.

Además, el conjunto de widgets debe ser representable, sin perder significado, en el mayor número de sistemas posibles. Esto significa que el número de widgets debe estar, a ser posible, cerca de la intersección de los que son representables en todas las arquitecturas y modos.

Nombre	Widget
but	botón
img	imagen (mapa de pixels)
gauge	barra de desplazamiento
panel	panel de texto
svg	gráficos vectoriales

Cuadro 4.1: *Conjunto de widgets del sistema*

Inspirándonos en otros toolkits y con la filosofía de que los widgets sean lo más versátiles que resulte posible hemos seleccionado este conjunto de widgets:

- **but**, botón.
- **img**, imagen.
- **gauge**, barra de control similar al control de volumen.
- **panel**, panel de texto con su barra de desplazamiento asociada y sus menús de edición.
- **svg**, gráficos vectoriales escalables.

Todos estos widgets mantienen la interacción confinada en la interfaz en la medida de lo posible. Por ejemplo, la edición de texto mediante el ratón, no se devuelve como interacción de bajo nivel, sino como eventos de escritura de cadenas de texto.

Estos elementos se completan con los metaelementos descritos anteriormente

- **row**, contenedor almacenador de elementos en fila.
- **col**, contenedor almacenador de elementos en columna.

Cada uno de estos elementos tiene un directorio asociado en el árbol de eventos en cuyo interior habrá un fichero **events** como se ha explicado anteriormente del que se leerán los eventos generados por el widget.

A continuación pasaremos a analizar cada uno de estos elementos y su interfaz concreto.

4.7.1. Botón

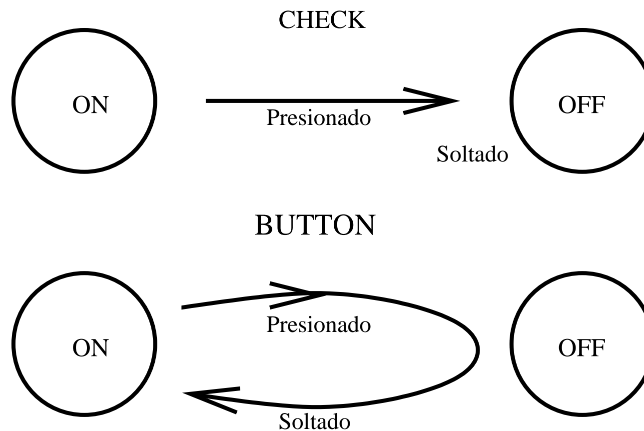

Figura 4.10: *Dos tipos diferentes de botones*

Un botón es un elemento que tiene dos estados, apretado y no apretado. Cómo se conmuta entre estos estados, está relacionado con el tipo del botón del que hablamos. Hay botones que vuelven al estado de no apretado automáticamente cuando el usuario deja de apretarlos. Hay otros, conocidos como check boxes o casillas, que conmutan de un estado al otro durante la interacción y luego conservan ese estado. Ambos tipos de botones se encuentran representados mediante este widget, que generará un evento con su nuevo estado cada vez que éste cambie.

El estado lo codificaremos como dos cadenas de texto, **on** y **off** que se podrán leer y escribir del botón **state**. El texto del botón será su nombre y el texto que se encuentre en éste con el carácter `'#'` representando un espacio y repetido dos veces para representarse a sí mismo. Esta estrategia de utilizar el carácter `'#'` sustituir al espacio se utilizará en general.

Otra parte del estado, opcional, es la utilización de una imagen para representar sobre el botón. Esta imagen se lee y se escribe como un JPG sobre el fichero **img**.

Finalmente, hay un fichero **ctl** en el que se pueden escribir o leer dos cadenas

de texto, **button** y **check**. Si el botón es de tipo button, sólo conmutará al estado contrario de aquel en el que se encuentre mientras el usuario lo presione. Si no, será permanente. Cómo funcionan ambos tipos de botones se puede ver en 4.10.

4.7.2. Imagen

El widget de imagen cumple una doble función. Por un lado sirve para representar imágenes en pantalla. Por otro exporta eventos de interacción mediante el ratón procesados de forma similar al modo cocinado de una consola de Unix [74] y eventos de teclado. Del sistema de cocinado del ratón hablaremos más en detalle en el capítulo de implementación. El estado de este widget consiste en una imagen. Esta imagen se lee y se escribe como un JPG o GIF sobre el fichero **img**.

El tamaño del widget dependerá de la implementación. Se utilizará, de forma similar a como se dijo con el botón, el nombre del fichero como representación alternativa de texto.

4.7.3. Barra de desplazamiento

La barra de desplazamiento, también llamada barra de control o de deslizamiento consiste en un widget que permite seleccionar un valor numérico en un rango de valores. Hay dos ficheros de control, **range** en el que se pueden escribir y leer el valor mínimo y máximo como cadenas de texto y otro fichero **value**, un fichero del que se puede leer el valor actual al que se encuentra la barra de control. Cada vez que cambia el valor, se genera un evento.

En la figura 4.11 se pueden ver una barra de control con dos valores diferentes de **range** y varias implementaciones de la barra de desplazamiento.

4.7.4. Panel

El panel de texto es el widget más complicado y completo. Realiza la función de permitir entrada y edición de texto. Esta entrada de texto se podría aislar

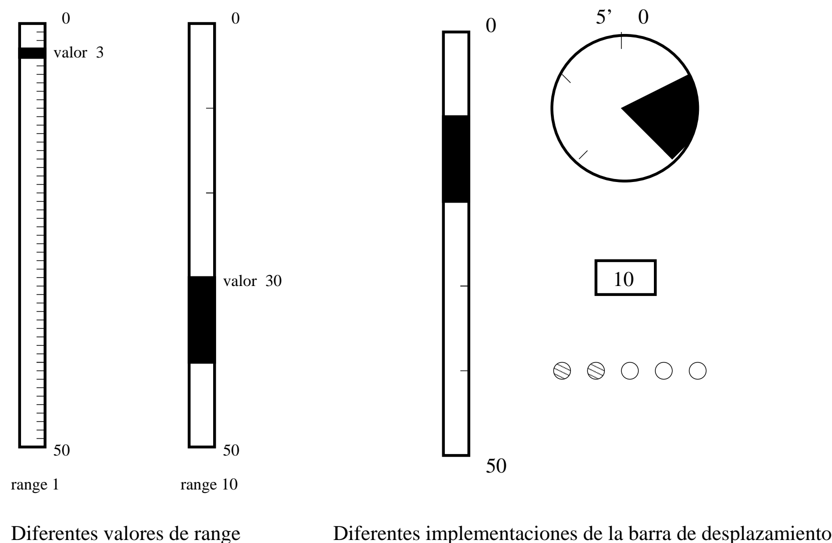

Figura 4.11: *Barra de desplazamiento*

completamente añadiendo sólo dos eventos, mandar el contenido del panel como evento y leerlo. El problema es que esto no resulta eficiente a la hora de actualizar las réplicas o de leer y escribir un fichero muy grande. En ambos casos son necesarias actualizaciones mediante eventos parciales de actualización del panel, añadir cadena y una dirección y borrar cadena con una dirección.

Por otro lado, hay otro tipo de eventos, como "última selección" y en general los eventos generados sobre el texto con el ratón. La última selección no se actualizará entre las vistas, pero si es necesario generar un evento para que la aplicación sepa cual es a la hora de ejecutar comandos sobre ella.

Este widget tiene un fichero **body** que contendrá el texto UTF8 del panel. Otro fichero **tag** contendrá el contenido de una barra horizontal opcional editable que se puede hacer aparecer o no en la parte superior del panel. El comportamiento de esta barra será similar al de la de acme [96].

Escribiendo en el fichero **ctl** se puede cambiar la aparición o no de la barra de deslizamiento asociada al panel. También se puede controlar la aparición de una barra horizontal editable o tag para guardar comandos. En este fichero se pueden escribir también comandos de edición de texto similares a los de ed o

sam [99] para controlar la selección. Un programa que implemente la lógica de aplicación, como puede ser un editor, puede utilizar este fichero, el de texto y la tag para controlar la interacción con el usuario durante la edición.

4.7.5. Gráficos vectoriales escalables

El widget de tipo **svg** contendrá un dibujo realizado mediante gráficos vectoriales escalables. Estos gráficos escalables permitirán dibujar figuras geométricas de diferentes colores como círculos, polígonos, curvas de Bezier, etc. Cada trazo tendrá una representación textual que se podrá leer y escribir del fichero **svg**, como se puede ver en la figura 4.12.

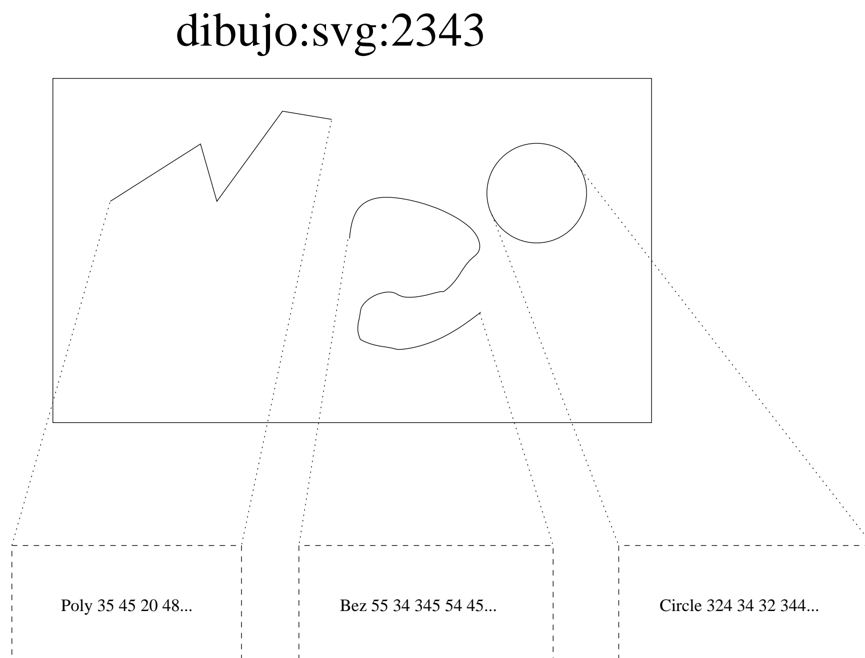

Figura 4.12: *Cada trazo se corresponde con una representación textual*

Las figuras se encontrarán representadas en una pantalla de alta resolución que se reescalará al tamaño adecuado según la implementación del servidor. Este widget generará trazos de ratón para la aplicación también en este espacio de coordenadas.

4.8. Contenedores

Los contenedores no son más que pistas para la implementación de qué tipo de relación semántica tienen los componentes entre sí pudiendo ser interpretadas o ignoradas adecuadamente si ello mejora el comportamiento en algún sistema.

Los tipos de elementos son:

- **row**, contenedor almacenador de elementos en fila.
- **col**, contenedor almacenador de elementos en columna.
- **win**, contenedor de alto nivel que representa una ventana y se comporta como una columna, sólo se puede crear en el directorio raíz.

Nombre	Elemento
row	Contenedor en fila
col	Contenedor en columna
win	Columna, ventana

Cuadro 4.2: *Tipos de metaelementos*

Estos metaelementos están representados por directorios en una estructura arbitrariamente recursiva que puede contener los directorios que representan a los widgets en cualquiera de sus niveles.

En el árbol de eventos, los contenedores son también directorios que contienen en su interior un fichero **events** cuya lectura representa interés en todos los eventos del subárbol que hay en su interior.

Para cambiar de tipo un contenedor hay que renombrarlo, lo que genera un evento de modificación en el contenedor del nivel superior, de forma que la aplicación y otras vistas se aperciban del cambio y actúen en consecuencia.

4.9. Codificación de los eventos

Como ya se ha justificado, los eventos representarán en la medida de lo posible acciones o sucesos de alto nivel, conteniendo en la interfaz la interacción de bajo nivel. Todos los eventos serán cadenas de texto.

En los pocos casos en los que los eventos pudieran necesitar ser binarios, como puede ser en un cambio de imagen en un widget que muestre una imagen, el evento será una notificación en forma de cadena de texto y un programa externo se ocupará de copiar la imagen. Esto mantendrá el flujo de eventos ligero y simple. Al tener eventos que se pueden incluir en un sólo mensaje, la interacción se hace más ágil, puesto que se reduce el número de rondas necesarias para las notificaciones y se sacan fuera del bucle los elementos más pesados.

4.9.1. Árbol de eventos

Los eventos se entregan a la aplicación mediante el sistema de ficheros utilizando unos ficheros especiales llamados **events** en el árbol de eventos. Estos ficheros se bloquearán en lectura hasta la aparición de un evento. La utilización de ficheros bloqueantes tiene la desventaja de que la aplicación debe tener un proceso para cada fichero en el que pueda quedarse potencialmente bloqueada o utilizar una estrategia similar a la llamada al sistema *select*.

Para que el número de procesos o de ficheros abiertos no crezca, es interesante utilizar algún sistema de agrupación de eventos, con ficheros de eventos que representen a grupos de fuentes de eventos.

Ambos diseños se pueden ver en la figura 4.13. Las lineas zigzagueantes representan procesos bloqueados en lectura en las cajas, que representan ficheros **events**. Si el número de ficheros creciese, se puede ver que en el caso de no agrupación de los eventos, el número de procesos crecería igualando la cantidad. En el sistema con agrupación, se tiene la ventaja de que se pueden agrupar eventos asociados a subárboles, pero sin perder granularidad. Esto sucede gracias a que se puede leer de un fichero más alto en la jerarquía (en la figura representado

por el fichero **events** del contenedor de tipo columna) o se puede leer de los ficheros de eventos asociados a los elementos de más bajo nivel.

A esto se debe añadir que también es conveniente tener un árbol separado del árbol de ficheros en el que se expone el estado de la interfaz de usuario con el fin de que esta se pueda copiar de forma sencilla con una herramienta general, sin tener que separar uno a uno los ficheros bloqueantes.

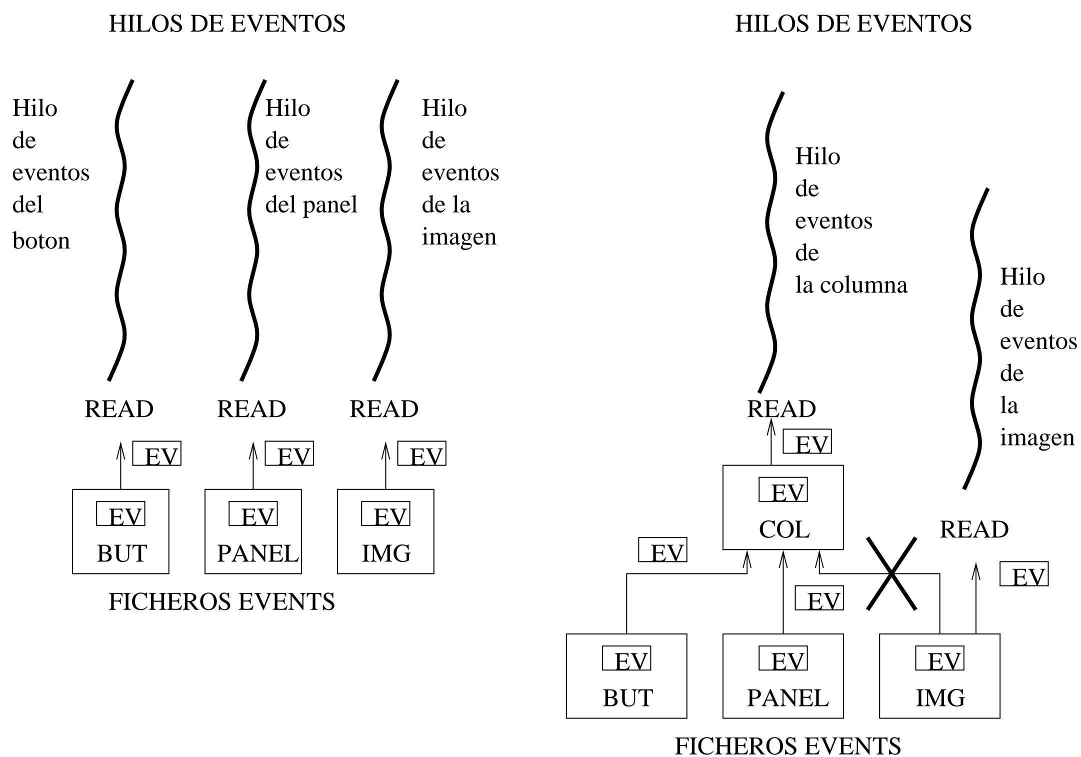

Figura 4.13: *Estructura plana versus jerarquía de eventos*

Como consecuencia de ambos requisitos, la mejor solución es una jerarquía de ficheros paralela a la jerarquía que contiene el estado. Un ejemplo de esto se puede ver en la figura 4.14. En nuestra arquitectura, Oni, esta jerarquía contiene en su interior directorios con el mismo nombre que la jerarquía que contiene el estado y dentro de cada directorio hay un fichero **events**. El árbol de directorios representa en ambas jerarquías la relación de inclusión o pertenencia a un grupo.

Habrá pues en el sistema de ficheros, dos árboles paralelos, uno para los eventos y otro para el estado de los widgets.

Los ficheros **events** en el subárbol de eventos, se utilizarán como ficheros bloqueantes de la forma descrita. Los ficheros **events** asociados a los directorios que representan contenedores, servirán como ficheros de agrupación y devolverán los eventos asociados a toda la subjerarquía incluida en el contenedor. Los eventos generados en un widget, subirán por la jerarquía de ficheros. Si en alguno de los ficheros hay un cliente bloqueado leyendo, el evento se drenará del sistema. Si no hay ningún cliente bloqueado en ninguno de los ficheros, el evento se guardará. En el momento en el que cualquier cliente lea de cualquier fichero por los que ha pasado el evento, se le entregará y se drenará del sistema.

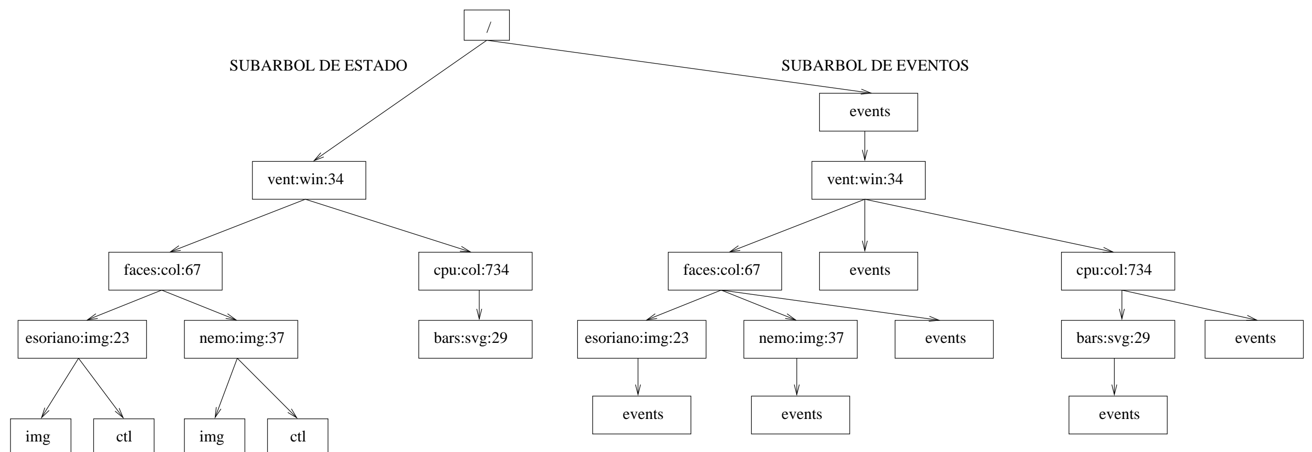

Figura 4.14: *Ejemplo de árboles de eventos y de estado*

En los ficheros events, se puede escribir para inyectar un evento al sistema y el efecto será el mismo que si el evento hubiese procedido de ese fichero aunque el estado de los widgets no cambiará, y habrá que actualizarlo de forma independiente. Aunque esto podría dejar a los widgets en un estado inconsistente con los eventos generados, como por ejemplo un botón puede generar un evento **on** sin cambiar de estado, esto permite depurar programas y el sistema de ficheros y generar eventos y cambios de estado automáticamente. Es suficiente

con ser cuidadoso con las actualizaciones y actualizar el estado para no entrar en estados inconsistentes.

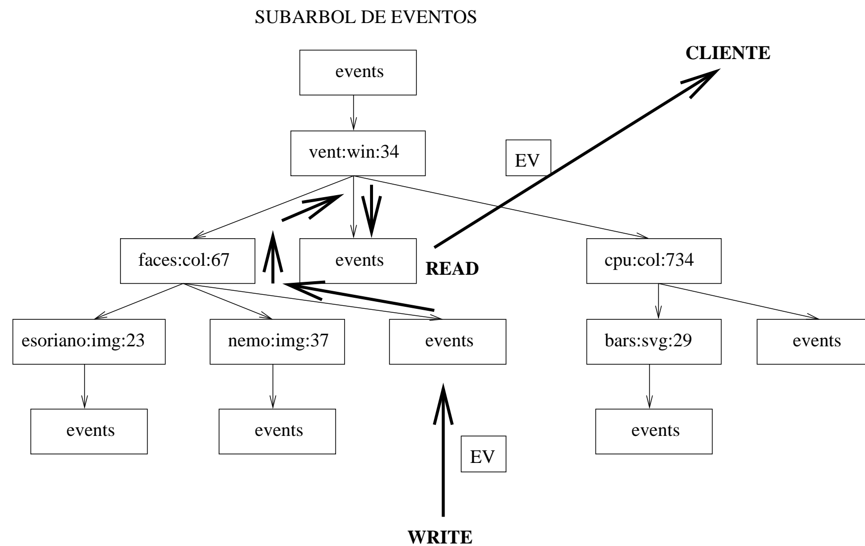

Figura 4.15: *Inyección de un evento en un fichero* **events**

4.9.2. Tipos de eventos: modificación e interacción

Hay que distinguir entre dos tipos diferentes de eventos, lo que llamaremos eventos de interacción y eventos de modificación. Los eventos de interacción los generará el usuario. Son fáciles de mantener a un alto nivel de abstracción, puesto que tienen un significado concreto que se puede capturar con el widget. Ejemplos de este tipo de eventos son presionar un botón o seleccionar un texto con el ratón.

Por otro lado están lo que llamaremos eventos de modificación. Los eventos de modificación son eventos necesarios para mantener a las vistas y al cliente informado de la interacción y de cambios en la interfaz. Son de más bajo nivel y su granularidad se establece mediante un compromiso entre latencia de modificación de las vistas y tráfico en la red. Algunos de estos eventos de modificación, como puede ser la aparición de una nueva vista o la conversión de una fila en

columna, no tienen un significado a alto nivel como el que se le puede conferir a los eventos de interacción. Las diferencias entre el procesado de ambos eventos se puede ver en la figura 4.16.

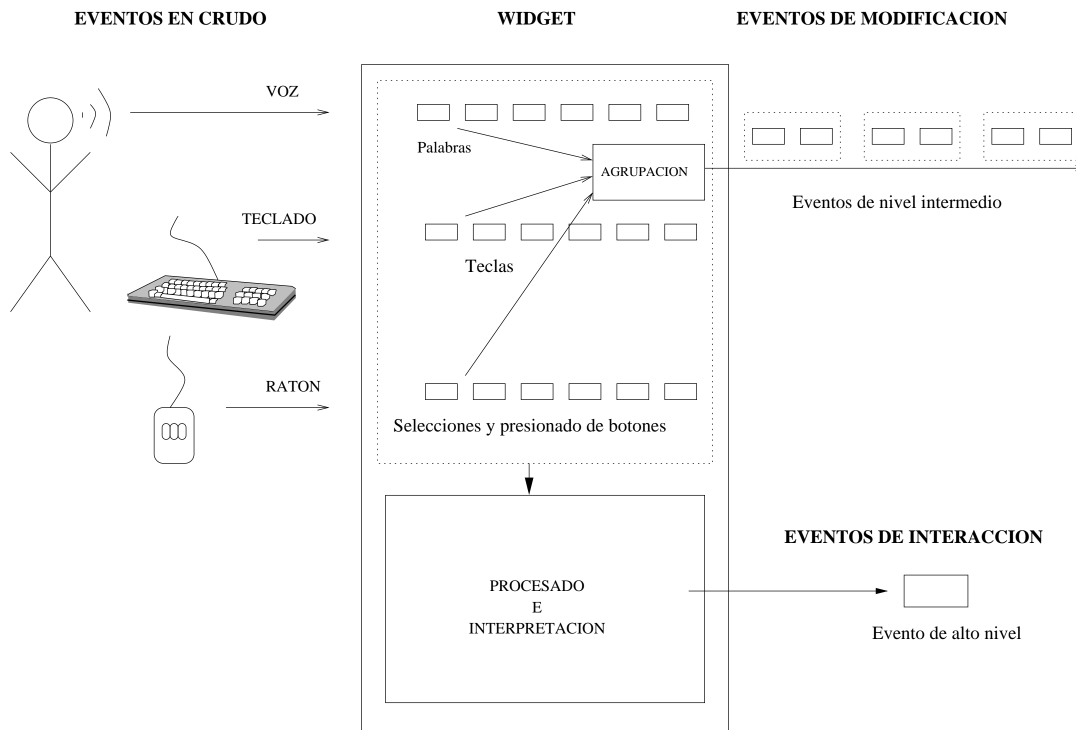

Figura 4.16: *Procesado de eventos de modificación y de interacción.*

Como ejemplo de las diferencias entre los dos tipos de eventos, vamos a analizar los compromisos existentes en los eventos que genera un panel de texto. Un panel de texto, es un rectángulo editable en el que se puede interactuar con el texto mediante el ratón o el teclado. Un ejemplo familiar de panel de texto sería el TextArea de HTML [30].

En un panel de texto tradicional, el programa responsable del panel de texto, que en el ejemplo sería el navegador web, tendría que manipular los eventos de ratón y de teclado, interpretarlos y modificar el texto adecuadamente. En nuestro caso, los eventos del ratón y del teclado se quedan confinados en el panel. El panel genera para la aplicación varios tipos de eventos de interacción y de modificación.

Los eventos de interacción que podría generar el panel podrían ser simplemente “se ha apretado el botón salvar”, o equivalentemente, “se ha seleccionado el texto de salvar y ejecutado” en una interfaz tipo typescript. Dicho de otra forma, se genera un único evento de botón apretado con el nombre “Salvar” y la aplicación se encargaría de leer el contenido del panel y escribirlo en un fichero.

Si sólo nos ocupamos de los eventos de interacción, basta simplemente, con generar eventos para la ejecución de comandos. Salvo por problemas de eficiencia en el manejo de paneles de texto asociados a ficheros grandes y condiciones de carrera entre los eventos de edición (contenidos en la interfaz) y los de ejecución de los comandos, con estos eventos bastaría.

Se podría generar también un evento con todo el contenido en texto del panel o con los cambios desde la última vez que se salvó. El nivel de abstracción al que se trabaja aquí es muy alto y en realidad se están interpretando las acciones del usuario a alto nivel, procesándolas y avisando a la aplicación de estas acciones ya procesadas.

Por otro lado, la posible existencia de vistas nos obliga a generar eventos para aquella parte del estado que queremos que se comparta entre las diferentes vistas. Hay partes del estado, como la selección del panel o el punto en el que se encuentra el cursor, que sencillamente podemos no querer que se comparta entre las diferentes vistas, lo que nos confiere más versatilidad en su uso. Sin embargo, el texto introducido por el usuario sí queremos que sea el mismo si queremos que los paneles se comporten realmente como vistas. Por ello habrá que generar cada cierto tiempo o número de eventos de ratón o de teclado un evento de modificación con la cadena de texto introducida o borrada. En un extremo, esto se realizaría cada vez que el usuario introduce un caracter o realiza una acción atómica con el ratón. En el otro extremo, se haría cada vez que el usuario pare de teclear o mueva el ratón. Existe, pues, un compromiso entre la sincronía de las vistas por un lado y el número de eventos generados y uso de la red por otro.

4.10. Interfaz de programación de los programas cliente

Los programas no necesitan manejar las vistas ni los directorios de forma explícita. Para ello utilizaremos una librería que realizará la función de abstracción y demultiplexación, como se puede ver en la figura 4.7.

Esta librería tendrá una interfaz similar a la de cualquier toolkit, con una pequeña diferencia. No se podrán usar atributos de bajo nivel, como el posicionamiento fino de elementos o si estos atributos se usan, serán ignorados o interpretados a los atributos de alto nivel que se exponen en el sistema de ficheros.

Los widgets elegidos son suficientemente de alto nivel y expresivos como para que pueda ser interesante escribir una implementación de esta librería que ofrezca la interfaz de programación de otra librería popular, como puede ser GTK [139]. Esto puede significar que puede ser necesario construir algunos de los widgets de GTK utilizando varios de los widgets de la arquitectura propuesta, Oni. Además, cierta información de posición, se perderá. Sin embargo, la consecuencia es que muchas de las aplicaciones pueden ser compiladas directamente contra nuestra librería. Aunque puede ser necesario retocar las aplicaciones si hay algún problema de usabilidad, sobre todo para las aplicaciones más complicadas, estamos hablando de cambios menores, y a cambio se obtienen todos los beneficios ya reiterados, de poder tener vistas, poder exportar el interfaz a la red, portabilidad y heterogeneidad.

Capítulo 5

Descripción de la implementación

Con el fin de validar la arquitectura se realizó una implementación sencilla de un prototipo de la misma. Esta implementación se realizó bajo Plan B [27], un sistema operativo derivado de Plan 9 [103]. Aunque el primer prototipo utilizaba elementos propios de Plan B, el segundo funcionaba sin modificación en Plan 9.

La implementación final se realizó utilizando un toolkit de widgets existente para Plan 9 llamado Control y una librería para la programación de sistemas de ficheros llamada 9P [79] [16].

5.1. 9P: Protocolo de Plan 9 para exportar árboles de ficheros y su librería

La librería 9P sirve para implementar servidores de ficheros mediante el protocolo 9P, un protocolo que sirve ficheros en base a RPCs con un interfaz bien definido.

9P está diseñado para exportar sistemas de ficheros representando recursos a través de un canal de comunicaciones. Estos sistemas de ficheros igual que en nuestro caso, suelen ser sintéticos, es decir, no representan ficheros respaldados por un disco y el protocolo está diseñado para que implementar este tipo de ficheros sea sencillo. A diferencia que otros protocolos como NFSv3 [37], 9P

tiene estado. Por cada conexión que realiza un cliente a un servidor, éste último mantiene un estado, como por ejemplo, qué clientes mantienen abiertos qué ficheros. Esto permite que 9P se pueda utilizar para recursos con un control de acceso sofisticado, como ficheros de apertura exclusiva y multiplexores de sesiones de chat.

9P es un protocolo basado en RPCs. Se basa en mensajes de petición, llamados mensajes T y mensajes de respuesta llamados mensajes R. Una RPC completa es un mensaje de petición y su mensaje de respuesta asociado. En 9P hay tres tipos de identificadores, representados por campos numéricos en el mensaje. El identificador principal se denomina *fid*. Este número representa un identificador de fichero y es unívoco por conexión y propuesto por el cliente. Representa un camino en la jerarquía de ficheros. Es similar a un descriptor de fichero en C, pero se puede utilizar para navegar por la jerarquía cambiando el camino que representa y además puede representar ficheros o directorios. Otro identificador importante recibe el nombre de *tag*. Este identificador lo elige el cliente y representa una RPC activa. Un mensaje T y uno R tienen el mismo tag. Mediante el uso de *fids* se puede multiplexar un canal de comunicaciones y mezclando diferentes RPCs distinguidas por los *tags*. El último identificador se llama *QID*. Un *QID* es un número que identifica unívocamente el objeto al que nos referimos dentro del servidor. Debido a que los ficheros se pueden reexportar y montar, un fichero puede aparecer varias veces en un sistema de ficheros. La forma de saber si es el mismo es utilizar su *QID*. De esta forma se identifica el fichero de forma completamente unívoca dentro de un servidor.

El conjunto de mensajes es muy pequeño, todos ellos salvo **Rerror** tienen un mensaje T y un mensaje R. Los mensajes son:

- **Version**: Negocia la versión del protocolo e inicializa la conexión.

- **Attach**: Presenta a un usuario al servidor, especificando además qué árbol requiere. Con este mensaje se inicia un montaje. El usuario se presenta al servidor y le dice en qué árbol de los que sirve esta interesado.

- **Walk**: Desciende por un nombre en la jerarquía de ficheros, para ello puede usar un fid o crear uno nuevo.

- **Clunk**: Borra un fid. Este mensaje libera todos los recursos asociados a un fid y permite la reutilización del identificador.

- **Error**: Sólo existe un mensaje R. Cualquier RPC que haya sufrido un error, recibe este mensaje de respuesta con una cadena de texto de error asociada.

- **Flush**: Aborta una RPC, identificada por la tag.

- **Open, create, read, write, remove, stat, wstat**: son las operaciones normales de un sistema de ficheros. Utilizan un fid para identificar el punto de la jerarquía en el que se encuentran y un QID en las respuestas para identificar unívocamente los ficheros y directorios.

En Plan 9 los servidores de ficheros se pueden escribir dentro del kernel, en cuyo caso, el protocolo 9P se realiza mediante llamadas a función. Los servidores se pueden encontrar también en espacio de usuario o en otra máquina a través de la red, en cuyo caso el servidor y el cliente se comunican mediante RPCs . Estas RPCs implementan las mismas funciones a las que llamaría el kernel si el servidor se encontrase en su interior. Esto se puede ver en la figura 5.1. 9P es una librería que se ocupa de implementar la serialización de las llamadas a función del protocolo 9P, de forma que sea sencillo atender a las RPCs que se realicen sobre nosotros. Básicamente, el programador rellena una tabla de punteros a función a los que se llama cuando llega una RPC. La serialización de los parámetros y de los valores de retorno los realiza la librería. Además, esta librería tiene soporte también para mantener actualizada de forma automática una estructura de datos en forma de árbol representando al árbol de ficheros y los metadatos.

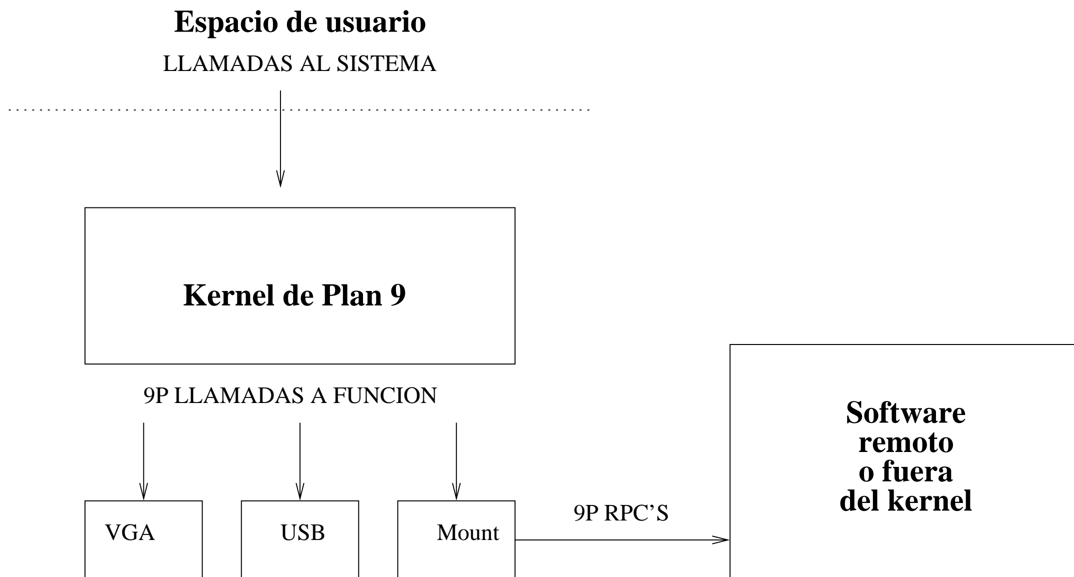

Figura 5.1: *9P como RPCs y como llamadas a función*

5.2. Librería de hilos de Plan 9

En Plan 9 hay una librería de hilos, *thread* [79] que hereda Plan B y que hemos utilizado para implementar el prototipo de esta tesis. Esta librería implementa las primitivas originales de CSP [68] con la adición de canales para anonimizar las comunicaciones.

En esta librería hay dos tipos de hilos, los *procs*, que son procesos expulsivos ligeros que comparten recursos y los *threads*, hilos colaborativos.

Los threads, que son el elemento básico de esta librería (cada proc ejecuta al menos un thread), se comunican mediante canales. Los canales son mecanismos de comunicación síncronos entre hilos. Sobre ellos se pueden realizar dos operaciones, mandar y recibir, bloqueantes ambas (a menos que haya búfering y el búfer no se encuentre lleno). Existe también una operación alt que permite recibir o mandar simultáneamente de varios canales. Al usar alt, el primer canal en mandar o recibir desbloquea la operación.

Los canales son el único mecanismo tanto de sincronización como de comu-

nicación existente en esta librería.

5.3. Control, librería de widgets de Plan 9

Control [79] es un toolkit de widgets gráficos propio de Plan 9. Se basa en un protocolo textual de eventos a través de un canal síncrono con un thread (hilo colaborativo de la librería *thread*) por widget. Los widgets de *control* tienen dos canales. En estos canales se escriben mensajes de texto con los atributos que se desean cambiar de los widgets.

Uno de los canales, el canal de control sirve para cambiar los atributos de los widgets. El otro canal es el canal de eventos y en él se reciben eventos transmitidos por los widgets.

Hay una función, **controlwire** que permite asociar varios widgets a un mismo canal de eventos o conectar el canal de eventos de un widget al canal de control de otro. Esta función es útil para crear widgets que sean el resultado de agregar varios widgets de control como se puede ver en la figura 5.2.

Control tiene un conjunto de widgets muy amplio que es un superconjunto del definido por Oni, lo que permite de forma sencilla implementar Oni basándose en Control. Control define también varios elementos agrupadores, dos de ellos fila y columna, idénticos a los de Oni y que se han utilizado para implementar éstos.

5.4. Evolución de los tres prototipos

Se han realizado tres prototipos. Es del tercer prototipo, que sirve ficheros y que separa el sistema de ficheros de la interfaz de usuario, del que hablaremos en este capítulo. No obstante, esta sección sitúa dicho prototipo en contexto.

- El primer prototipo es un programa monolítico que integraba el sistema de ficheros y la interfaz de usuario.

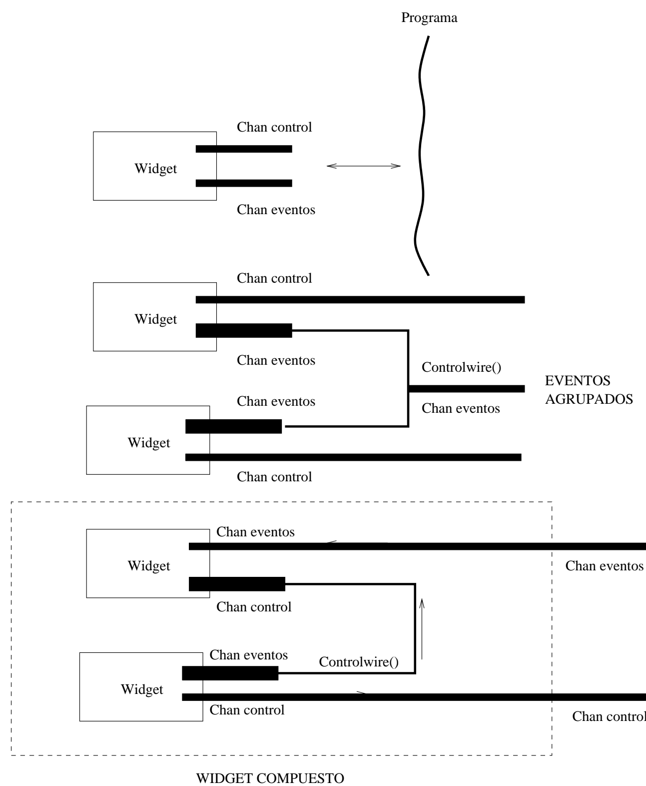

Figura 5.2: *Widgets de control y uso de controlwire()*

El programa se hizo enormemente complejo y las condiciones de carrera lo complicaron enormemente. Estas condiciones de carrera eran consecuencia de que hay varios puntos de entrada de datos. Por un lado está el sistema de usuario y por otro los widgets y ambos son asíncronos.

Otro detalle importante es que en el primer prototipo el sistema de ficheros es en realidad un sistema de cajas [51], una abstracción similar a la de fichero pero que sustituye las operaciones de lectura y escritura por una sola de copia. Esta abstracción tiene la ventaja adicional de que una caja se podía listar o leer su contenido, actuando como directorio y fichero al mismo tiempo. Sin embargo, esta abstracción se abandonó en nuestro sistema, Plan B, debido a la complejidad que suponía reescribir todas las aplicaciones para que utilizasen cajas. Cada uno de los programas tenían

que ser reescrito para utilizar cajas, incluyendo programas básicos como ls y tar.

Esto ralentizaba enormemente el desarrollo y como consecuencia, se decidió abandonar las cajas, a pesar de sus ventajas y tratar de integrar éstas en la medida de lo posible dentro del sistema de ficheros. Esto tuvo como consecuencia el desarrollo de un segundo prototipo.

La escritura de un sistema de ficheros de widgets es complicada por la entrada de dos tipos de eventos asíncronos, los que proceden del sistema de ficheros a través de las RPCs de 9P y los que proceden de la interfaz de usuario de los widgets de Control. Es sencillo que se produzcan problemas de concurrencia o condiciones de carrera.

- El segundo prototipo era un programa monolítico en el que no había suficiente separación entre el sistema de ficheros y la interfaz de usuario. Este programa se hizo muy complejo de depurar por problemas de concurrencia.

- Como consecuencia, se reescribió completamente un tercer prototipo comenzando desde cero. En esta versión la interfaz de usuario estaba completamente separada del sistema de ficheros, lo que simplifica la depuración.

  Con el fin de hacer esta separación completa, el programa del tercer prototipo se compone de dos procesos diferentes comunicados por pipes. Al estar ambos procesos en espacios de memoria independientes y tener pipes de comunicación que actúan como canales síncronos entre ellos aislándolos resultó mucho mas fácil depurarlos.

  Uno de los problemas adicionales que causaron que esta separación fuese necesaria es la forma en la que se suelen construir las aplicaciones en Plan 9. Las aplicaciones en Plan 9 se ejecutan sobre una sola ventana. Como consecuencia, muchas de las variables asociadas a la ventana son variables globales cuando se utiliza la librería estándar de dibujo de Plan 9. Aunque hay formas de crear y mantener estructuras de datos nuevas y

separadas para cada ventana, se complica enormemente la programación. El ejecutar un proceso separado por ventana, hace esta separación trivial.

Una ventaja adicional de esta separación es que es más sencillo trazar qué esta sucediendo mediante el uso de pipes al tener un bus de comunicación que obliga a serializar todas las operaciones, lo que permite imprimirlas.

5.5. Concurrencia, sistema de ficheros, interfaz de usuario

La arquitectura propuesta, Oni, se compone de dos programas diferentes: el sistema de ficheros, *uifs*, y un programa que se ejecuta por cada ventana, *9win*.

Uifs es un programa único (aunque compuesto de varios procs), que representa la interacción con el sistema de ficheros. En *uifs* se encuentra el programa que atiende las RPCs de 9P. Uifs no contiene ningún estado propio, salvo representaciones del estado de *9win* que consulta para cada operación.

9win representa el estado de una ventana y por tanto, hay uno por ventana. Contiene en su interior los widgets de *Control* y se ocupa de la interacción con el usuario.

Ambos procesos se comunican para notificarse cualquier cambio que se realice en el estado de los widgets. Sin embargo, el estado de los widgets se centraliza en *9win*.

Uifs se comunica con *9win* mediante RPC's sobre dos pipes, uno para eventos y otro para operaciones sobre los widgets. Cada uno de estos programas está formado por varios procs de la librería *thread* comunicados por canales.

Cada widget está representado en ambos programas, en *9win* por el propio widget y en el sistema de ficheros por un manejador o proxy (que llamamos gproxy). Este gproxy se utiliza comunicarse con la implementación concreta. Para aislar la implementación concreta de cada widget, el proxy contiene una tabla de punteros a función. Estas funciones realizan RPCs de forma opaca al

que llama a la función. La arquitectura de esta implementación se puede ver en la figura 5.3.

Los cambios o peticiones en los widgets se pueden hacer a través de dos interfaces diferentes con el usuario, el sistema de ficheros y la interfaz de usuario. Las operaciones sobre el sistema de ficheros se traducen en RPCs sobre el pipe de operaciones, utilizando los gproxies como manejadores. En los gproxies se encuentran los identificadores necesarios para hacer estas operaciones unívocas. Las operaciones sobre la interfaz de usuario, generan eventos que se transmiten a través el pipe de eventos.

Los hilos utilizados en el prototipo son todos procs, es decir procesos ligeros expulsivos planificados por el kernel. Cada proc se ocupa sólo de una actividad y se comunica con los del mismo dominio de memoria o programa mediante chans. Hay dos programas *uifs* y *9win*. En *uifs* hay cuatro procesos, pipeevproc, evproc, uifsproc, listener y srv. Pipeevproc lee del pipe de eventos, desempaqueta el mensaje a una estructura y lo escribe en el canal evchan. Evproc es el proceso que se ocupa de atender los eventos, actualizar lo que sea necesario en las estructuras de datos de los proxies y del sistema de ficheros, ver si hay clientes esperando para recogerlos y si no guardarlos hasta que los haya. Listener se ocupa de escuchar esperando si hay nuevas conexiones en la red. Cuando llega una nueva conexión, crea un hilo srv para atender a los clientes del sistema de ficheros. Uifsproc es el hilo que se encarga de hablar con las interfaces de usuario a través del pipe de operaciones para actualizar los cambios que se produzcan desde el sistema de ficheros.

En *9win* hay otros cuatro procesos, opdispatch, evsender, winkiller y mousekbd. Mouse y kbd son threads que arranca control que se encargan de recoger los eventos de la interfaz de usuario y de mandarlos a través del canal de eventos. Evsender serializa estos mensajes y los manda a través del pipe. Opdispatch ejecuta las operaciones que llegan a través del pipe de operaciones. Winkiller hace polling del directorio de rio (el sistema de ventanas) asociado a la ventana y en caso de desaparición mata todos los threads de *9win* (que ha su vez manda EOF

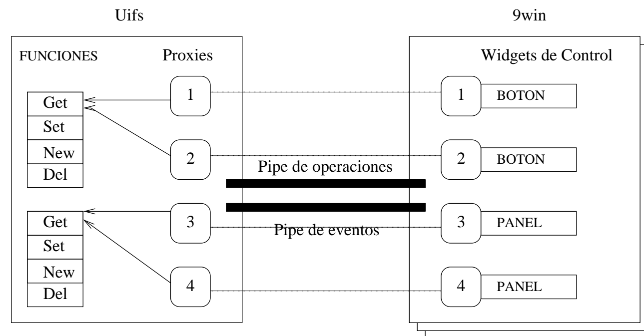

Figura 5.3: *Estructura de la implementación*

a través de los pipes) de forma que se pueda hacer recolección de basura.

5.6. Protocolo de mensajes en los pipes

A continuación explicamos el protocolo de RPCs que se habla sobre los pipes, tanto de operaciones, como de eventos. En este protocolo existen dos tipos de RPCs. La RPC **Op** va de *uifs* a *9win* y sirve para cambiar el estado de un widget. La RPC **Ev** va de *uifs* a *9win* y sirve para transmitir eventos. Hay dos pipes y cada uno se utiliza para transmitir los mensajes asociados a una de las RPCs. El protocolo en total se compone de dos tipos de mensajes, mensaje de operación y mensaje de eventos asociados a las RPCs y que se transmiten uno en cada pipe y un mensaje de respuesta puramente textual que permite retornar la respuesta a la RPC y que es igual para ambas.

Los campos de los mensajes se pueden ver en la figura 5.5 y los valores de cada campo se pueden ver en las tablas 5.1 y 5.2

En los mensajes de tipo operación, que se mandarán a través del pipe de operaciones, Op, como se puede ver en la figura 5.5, **Op** representa el código de la operación, que puede ser **Newel, Deleteel, Getdata** o **Setdata**, que crean un elemento, lo borran, extraen su estado o escriben su estado respectivamente. **Ctnup** es el identificador del contenedor del elemento. **Id** es el identificador del elemento, un número único en el ámbito de una ventana. Type es el tipo de

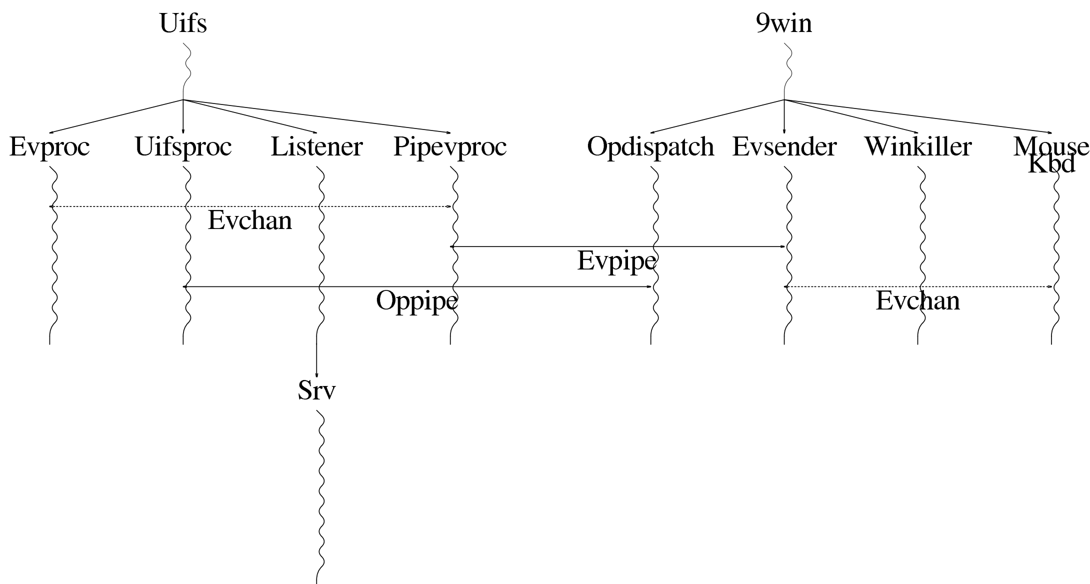

Figura 5.4: *Threads existentes en la implementación*

elemento, **Trow, Tcol, Tbut, Tpanel, Timg, Tgauge** o **Tsvg**. **Datasz** es la longitud del campo de datos que contiene datos para los componentes. El campo de datos **Data** es un campo opaco para todos los intermediarios y que debe ser tratado como datos binarios. Contiene datos que sólo el componente debe interpretar y que representarán su estado. Los errores en el estado se tratarán en el componente y se devolverá un código textual en la respuesta.

En los mensajes de tipo evento, que se mandarán a través del pipe de eventos, Ev, hay tres campos. El primer identificador **Wid** identifica al proceso del que viene el evento. Aunque se puede distinguir por el pipe del que viene el mensaje, resultaba más cómodo implementarlo así ya que es muy sencillo conseguir un identificador único para el *9win* utilizando el identificador del proceso. Además esto permite depurar más fácilmente, puesto que con ver una traza de los mensajes ya se sabe que está sucediendo. Este identificador, concatenado al **Id**, que es un identificador único local al *9win* construyen un identificador unívoco para el widget.

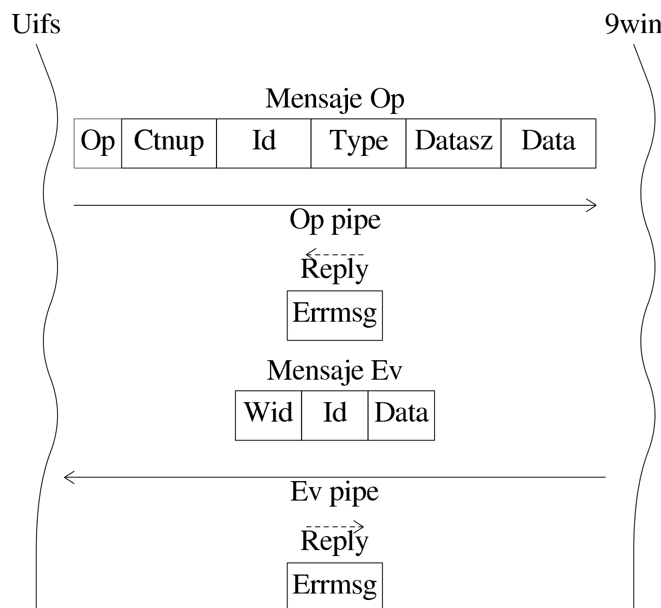

Figura 5.5: *Protocolo de los pipes existentes en la implementación*

5.7. Sistema de ficheros

Implementamos un subconjunto de las operaciones del sistema de ficheros y para el resto utilizamos funciones genéricas de las que nos provee la librería de 9P. Las operaciones que implementamos fueron *create, read* y *write*. Más adelante implementamos *flush* para que los clientes pudiesen interrumpir operaciones. Después añadimos también *attach* y *read* y *write* modificados para la autenticación de clientes.

La operación más compleja es la de crear un fichero o directorio. Primero se procesa el nombre del fichero. Después, se realizan diferentes acciones dependiendo del tipo del fichero creado. Si es un contenedor y se encuentra en el nivel más alto de la jerarquía se crea una ventana que es la forma de agrupar mas alta de la jerarquía del sistema de ventanas. A continuación se actualizan los dos árboles, el de eventos y el de contenidos y se generan los eventos de actualización necesarios.

Un problema que no pudimos prever es que hay que tener cuidado en el orden

Nombre	Significado	Valores
Op	Código de la operación	**Newel, Deleteel, Getdata, Setdata**
Ctnup	Id contenedor	Número único por ventana
Id	Id widget	Número único por ventana
Type	Tipo del elemento	**Trow, Tcol, Tbut**
		Tpanel, Timg, Tgauge, Tsvg
Datasz	Tamaño en bytes del campo Data	Numérico
Data	Campo Data	Bytes opacos

Cuadro 5.1: *Mensaje* **Ev**

Nombre	Significado	Valores
Win	Ventana a la que pertenece	Número único
Id	Identificador del widget que generó el evento	Número único por ventana
Data	Datos del evento	Bytes opacos

Cuadro 5.2: *Mensaje* **Op**

de las operaciones realizadas simultáneamente a través de la interfaz de usuario y del sistema de ficheros. Un ejemplo de esto es cuando un cliente cierra una ventana. Si esto sucede a la vez que una operación se realiza sobre el sistema de ficheros en un elemento en el interior de la ventana, se puede llegar a una situación peligrosa. El uso de RPCs sobre pipes simplificó mucho estas operaciones, porque separa la concurrencia sobre el sistema de ficheros y sobre la interfaz de usuario. El orden en las operaciones, es primero actualizar las variables locales y luego realizar RPCs a través de pipes. Estas RPCs pueden devolver errores si ha desaparecido el elemento o EOF si se ha cerrado el pipe porque ha desaparecido la ventana. De esta forma, está mas claro en que estado se encuentra cada operación en cada momento y cómo recuperarse de un error.

5.8. Utilización de control

Los elementos gráficos que utilizamos, salvo el **panel** eran los que provee control. Los widgets y contenedores en *9win* están representados mediante una

estructura de datos llamada gelem que se encuentra representada por gproxy en *uifs* ifs. Estos elementos tienen, igual que gproxy una serie de punteros a función que abstraen del elemento concreto al que se refieren las operaciones y que están estrechamente relacionadas con las operaciones definidas en el protocolo de pipes. Estas funciones son **newgel, stickgel, setdata** y **deletegel**. Si el widget del que tratamos se encuentra ya en libcontrol, no es necesario implementar todas las funciones, ya que algunas de ellas son genéricas.

Newgel crea un nuevo elemento utilizando los datos contenidos en la estructura de datos **gelem**. **Stickgel** lo añade a la jerarquía de elementos, haciéndolo activo o visible. **Setdata** actualiza los datos correspondientes en la estructura y las estructuras de datos asociadas. No existe una función **getdata** correspondiente porque los datos tienen que estar antes en la estructura, con lo que no es necesario acceder al widget para extraer los datos. **Deletegel** desactiva el elemento, lo extrae de la jerarquía y lo destruye.

La primera parte de la creación de un widget, en la operación **opdispatch** que se encarga de atender a las operaciones que provienen del sistema de ficheros, rellena los punteros a función de **gelem** dependiendo del tipo del mensaje. Después, todas las operaciones se hacen de forma genérica usando estos punteros.

Es sencillo añadir un widget nuevo sin utilizar control. Basta con implementar las cuatro funciones descritas anteriormente y generar los eventos en el mismo formato que control a través de un canal. Todo el sistema está diseñado para que se integre perfectamente.

5.9. Minilenguaje de uso del ratón

Control no contenía un panel de texto como parte de su conjunto de widgets. La implementación de este panel resultó ser compleja al ser un widget tan versátil y con un lenguaje de interacción tan complicado. Con el fin de simplificar esta implementación se escribió un sistema de cocinado del ratón [86]. Este sistema evolucionó y se convirtió en un minilenguaje de procesado del ratón.

La idea de este minilenguaje es que permite describir una forma de interacción con el ratón. En base a esa descripción o programa, se procesan los eventos en crudo generando eventos cocinados que representan la forma de interacción descrita. Esto se puede hacer de forma global, definiéndose un lenguaje que defina completamente la interacción con el ratón o de forma local a cada widget.

Ha habido dos versiones de este sistema. Una primera versión, cocinaba los eventos del ratón de forma transparente para las aplicaciones que no los usaban. Estas aplicaciones podían enlazarse con ella y utilizar o no los eventos cocinados, que eran una adición de algunos bits al mensaje estándar que devolvía la librería del ratón. Esta versión utilizaba un minilenguaje propio, V1 que describiremos a continuación. Tenía dos problemas. El primero es que el procesado del minilenguaje se complicaba en exceso a pesar de ser muy sencillo. Además al lenguaje le faltaba expresividad lo que, como veremos, hacía los programas demasiado largos y difíciles de depurar. El segundo problema, es que la aproximación de modificar los mensajes del ratón es errónea. Para que el sistema se pueda apilar y sea más limpio el modelo de programación, es mejor servir eventos de ratón procesados en un nuevo canal (el mecanismo de comunicación de la librería de threads) o varios.

La segunda versión, se escribió utilizando la librería regexp [79], lo que simplificó enormemente el parsing y permitió ampliar de forma sencilla la capacidad expresiva del lenguaje. Además la segunda implementación sirve dos canales, uno con los eventos sin procesar y otro con los eventos cocinados. Estos eventos están escritos en un formato tal que se pueden "recocinar" si fuese necesario.

Detallaremos primero la versión original del lenguaje para que se puedan ver los problemas que produjo y luego la segunda con las diferencias y soluciones a los mismos.

5.9.1. Primer prototipo

La arquitectura general del primer prototipo se puede ver en la figura 5.6. Los mensajes generados por el ratón se tokenizan y se generan caracteres repre-

sentando a estos eventos.

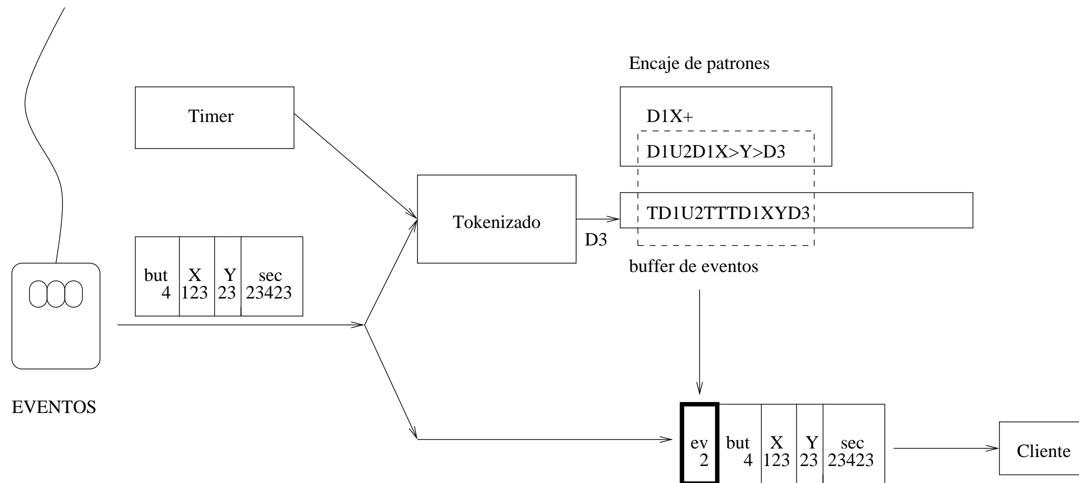

Figura 5.6: *Estructura del procesado de V1*

Tokenización

La tokenización es extremadamente sencilla. Basta con comparar el mensaje de ratón con el último mensaje recibido.

- Un movimiento de un cuanto en el eje x genera el carácter 'X'
- Un movimiento de un cuanto en el eje y genera el carácter 'Y'
- Presionar el botón **N** genera la cadena **DN** , por ejemplo **D2**
- Soltar el botón **N** genera la cadena **UN** , por ejemplo **U1**
- Cada cuanto de tiempo, un timer genera **T**

Como resultado de la tokenización, se generan tokens, representados por cadenas de caracteres que se van acumulando en un buffer. Con la llegada de un nuevo token, el contenido del buffer se compara con una serie de patrones que forman el programa para interpretar el ratón. Según sea el resultado de esta

comparación, se marca el mensaje de ratón con los bits adecuados y se drena o no la parte del buffer que ha encajado con el patrón.

Un programa consiste en patrones y a cada patrón hay asociado un conjunto de bits o banderas que son las que se marcan en el mensaje del ratón y un conjunto de acciones asociadas, como drenar o no el patrón del buffer. Las acciones se aplican y los bits se ponen a uno cuando el patrón encaja con un sufijo del buffer.

Un problema es que pueden quedar sufijos de acciones en el buffer que no encajen con ningún patrón y hagan que el buffer se llene. Con este fin, el buffer también se drena cuando el patrón no encaja como prefijo de ninguno de los patrones definidos o cuando se llena. Una regla que hay que añadir y que es obligatoria es que una cantidad suficientemente grande de cuantos de tiempo solos, como por ejemplo veinte, drenan el buffer. Esto hace que el buffer no se llene de T's y las acciones reales no quepan en él. Por supuesto el buffer es suficientemente grande.

Metalenguaje V1 de descripción de patrones

Con el fin de describir los patrones con los que debe encajar el resultado de la tokenización en el buffer, diseñamos el metalenguaje V1, un lenguaje para describir un conjunto de gramáticas que es un subconjunto de los expresables mediante expresiones regulares. Como se demostrará más adelante este lenguaje es expresable en base a expresiones regulares lo que no significa que describa lenguajes tipo 3 de la jerarquía de Chomsky ya que hay variables que son equivalentes a las referencias a expresiones parentizadas que también existen en las expresiones regulares estándar de Unix [52].

Hay cuatro tipos de cadenas en el metalenguaje: literales, múltiples, variables y repetidas.

Las cadenas literales, son simplemente cadenas de tokens no incluidos en los demás tipos de cadenas y encajan consigo mismas. Son los tokens asociados a caracteres numéricos y las letras mayúsculas excluidas la **M** y la **N**. Dicho de

otra forma, **X, Y, T, D, U, 1, 2, 3, 4, 5, 6, 7, 8 ,9, 0** encajan consigo mismos.

Las cadenas múltiples son cadenas que representan a varios tokens. Estas son:

- **M** encaja con **X** ó **Y**
- **N** encaja con **1, 2, 3, 4, 5, 6, 7, 8, 9, 0**

Las variables, toman un valor la primera vez que encajan con un sólo carácter numérico y luego solo encajan con ese valor. Están representadas por letras minúsculas. Por ejemplo, **DiTTTUi** encaja con **D1TTTU1** pero no con **D1TTTU2**. Cuando estas variables encajan, además se guarda el valor de la variable, que luego se devuelve como botón presionado en el mensaje de ratón.

Las cadenas repetidas son cadenas de varias cadenas literales iguales seguidas del carácter ‘>’ o‘<’

- N literales de un tipo seguidos de ‘>’ encajan con al menos N literales del mismo tipo.
- N literales de un tipo seguidos de ‘<’ encajan con como mucho N literales.

Un ejemplo de patrón sería, por ejemplo, **D1U1TT<D1**. Este patrón es el patrón asociado al doble click en el botón 1. Encaja con **D1U1D1** ó **D1U1TD1** ó **D1U1TTD1**. Estas cadenas significan que se presiona el botón 1, se suelta, se espera como mucho un tiempo menor que dos veces el cuanto de tiempo y luego se presiona el mismo botón de nuevo.

Este lenguaje es bastante expresivo, pero no lo suficiente. Tiene tres problemas fundamentales:

- Hay reglas en las que da igual el orden. **D1D2U2U1** y **D1D2U1U2** pueden ser equivalentes para nosotros. Para expresar esto en V1 es necesario utilizar varias reglas que incluyan todas las combinaciones posibles. Esto es tedioso y es fácil cometer errores.

- En ocasiones es cómodo ignorar algunos tokens para algunas reglas. Por ejemplo, un doble click podría ser **D1U1TT<D1** ó **D1MU1TT<D1** ó **D1U1MTT<D1** ó **D1U1T<MT<D1** ó **D1U1TT<MD1**. Sería interesante simplemente ignorar **M**

- A veces es interesante utilizar‘>’ sobre una subcadena completa. Para ello es necesario algún tipo de agrupación, por ejemplo utilizando paréntesis, pero eso complica el parsing enormemente.

Sería interesante definirse un lenguaje nuevo o al menos extensiones del mismo que simplificasen la programación, no tuviesen los problemas comentados y fuese más sencillo de programar. Para eso diseñamos el lenguaje V2. Contiene algunas extensiones a la definición del lenguaje original y además está implementado mediante el uso de expresiones regulares, con toda su capacidad expresiva. Además el uso de la librería Regexp [79] simplifica enormemente el encaje de los patrones.

Lenguaje V2 de descripción de patrones

El lenguaje V2 es un superconjunto del lenguaje V1 con un cambio fundamental en la implementación y es que se traduce a expresiones regulares, lo que significa que además de usar el lenguaje V2, tenemos toda la potencia de las expresiones regulares. Esto resuelve dos problemas, por un lado hace al lenguaje más expresivo y por otro hace el parsing más sencillo, ya que se puede usar la librería de expresiones regulares. Se han añadido dos extensiones al lenguaje, las variables de conjunto y el anulado de tokens, ambas para resolver los problemas que apuntábamos anteriormente que no son fácilmente solubles mediante el uso de expresiones regulares.

El anulado de tokens significa simplemente que para cada patrón hay un conjunto de tokens que indica que tokens del buffer que se borran o ignoran antes de realizar el encaje.

Las variables de conjunto significan que las variables separadas por el carácter `'o'` encajan con el conjunto de valores de cada una de las variables, siendo un error el que no se encuentren ya instanciadas. Por ejemplo si: $i = 1$ y $j = 2$ **ioj** encaja con **1** o con **2**.

Aunque las variables, incluso las de conjunto, se podrían expresar mediante las referencias a expresiones parentizadas de las expresiones regulares, es mucho más sencillo programarlas por separado, comprobando que encajan y sustituyéndolas por su valor antes del encaje de patrones.

Como ya se ha explicado, las expresiones en el nuevo lenguaje, ahora se sustituyen por expresiones regulares. Esto se puede hacer en tiempo de compilación, ya que las reglas son conocidas a priori. Esta traducción la vamos a expresar mediante expresiones regulares.

- El patrón `([^DUT]*[<>]?)[^T])` se sustituye por `\1T*\2`
- El patrón `>` se sustituye por `+`
- El patrón `([09TXYNM])+<-` se sustituye por `\1?` repetido el mismo número de veces. No se puede usar una sola expresión regular porque en el patrón de encaje no se puede hacer referencia a patrones, lo que haría que el patrón sencillamente fuese `([0-9TXYNM])\1<` y se sustituyese por `\1<\1?`. Por ello es necesario utilizar expresiones regulares estructuradas [98], de forma que primero se encaja con `([0-9TXYNM])<` y luego se sustituye `[0-9TXYNM]` por `\1?`.
- El patrón `N` se sustituye por `[0-9]`
- El patrón `M` se sustituye por `X|Y`

La figura completa del funcionamiento del procesado de eventos usando V2 se puede ver en la figura 5.7. Aunque parece más complicada que el procesado mostrado en la figura 5.6 en realidad no lo es, ya que el encaje de patrones aquí es trivial, pues se realiza mediante la librería regexp. Además la sustitución

de variables, que aparece aquí de forma explícita, ya se hacía como parte del encaje de patrones, pero ahora se ha separado para poder utilizar la librería de expresiones regulares directamente. Esta separación no es completa. Si la sustitución de variables se hace en el bucle de encaje de patrones, es más sencilla de implementar. En caso contrario, habría que iterar en los caracteres que indican repetición respetando los paréntesis para hacer la sustitución de variables. Hemos tomado la primera opción.

El recuadro punteado, es la traducción a expresiones regulares ya descrita.

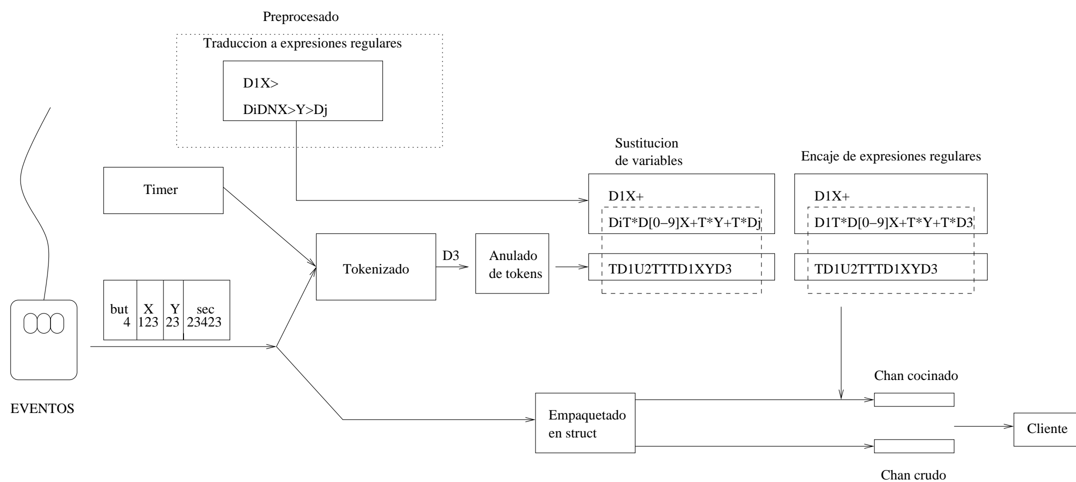

Figura 5.7: *Estructura del segundo procesado de V2*

A continuación se presenta un ejemplo de programa en el lenguaje V2 representando el lenguaje de interacción de ratón de un editor como acme [96]. La E mayúscula significa que el patrón que encaja se extrae de la pila de eventos, mientras que la P significa que sólo se emite un evento cocinado, pero no se extrae el patrón de la pila.

```
Largo rato: TTTTTTT> E
Mover: M E
Click: Di E
Doble click selección: DiUiTT<Di P
```

```
Doble click: DiUiTT<DiUi E Ignorar: M
Selección: DiMM> P Ignorar: M
Chord: Di(DjokUjok)*Djok P Ignorar: M
Chord end:  Di(DjokUjok)*DjokUjok E Ignorar: M
```

Capítulo 6

Evaluación de la arquitectura

La arquitectura propuesta resuelve los problemas que se establecieron en la introducción mediante el uso de un sistema de ficheros. Cómo se ha explicado a lo largo de todo el presente documento, la combinación de una interfaz uniforme con un modelo bien conocido, junto con el sistema de nombrado jerárquico que suministra y la exportación de objetos de más alto nivel que en los modelos tradicionales, (widgets) hacen que Oni pueda resolver de forma sencilla y portable los problemas de mantenimiento transparente de vistas, de adaptación, de protección y transparencia de localización.

6.1. Heterogeneidad, diferencia de capacidades y portabilidad

Prácticamente todos los sistemas operativos y muchos runtimes tienen capacidades de compartición de ficheros. Esto hace que sea extremadamente portable el uso de un sistema de ficheros como interfaz de comunicación entre la interfaz de usuario y la aplicación. Además, casi todos los lenguajes de programación tienen mecanismos para leer y escribir ficheros.

En Plan 9 existen multitud de servidores para los sistemas de ficheros en red más utilizados, como puede ser un servidor de CIFS [14] o uno de NFS

llamado nfsserver [79]. Esto hay que unirlo a la flexibilidad de sus espacios privados de nombres [104], y a la capacidad de anunciar árboles de ficheros que añade Plan B [27]. Gracias a todas estas capacidades es muy sencillo construir pasarelas para la mayor parte de los sistemas de ficheros existentes que anuncien subárboles de la interfaz de usuario que pueden cambiar de forma dinámica.

La única preocupación que podría surgir es si se puede ejecutar Oni en un sistema que tenga poca capacidad computacional. Por un lado, el interfaz que exporta Oni, es de muy alto nivel, lo que significa que no requiere cálculos complicados que precisen de coma flotante. Por otro lado es sencillo hacer un sistema de ficheros que ejecute en un sistema con pocas capacidades. Un ejemplo de esto es [82], un sistema que exportaba robots de lego mediante un protocolo llamado styx [105] un pariente muy cercano de 9P, de hecho, prácticamente idéntico salvo por la autenticación.

Otros sistemas, como puede ser Fresco [109] o Gnome [111] utilizan o han utilizado CORBA [149] o DCOM [63] en el lugar donde nosotros utilizamos un sistema de ficheros. El servidor de nombres de ambos nos provee de una estructura jerárquica que se expone a la red de forma similar a un sistema de ficheros y en ambos existen modelos y arquitecturas para proveer seguridad y algunos otros de los requisitos que hemos expuesto aquí, se seguridad, y reflexión. Sin embargo ambos fallan en portabilidad por varias razones. La primera y más importante es su complejidad. La complejidad de los mecanismos utilizados hace que sea difícil escribir el software necesario, que sea lento y que no pueda ejecutar en sistemas con limitaciones computacionales. Esto solamente, ya ha causado que muchos de los proyectos utilizando CORBA lo abandonen por subconjuntos de los mecanismos de CORBA implementados de forma más sencilla. Un ejemplo de esto es KDE [135].

La segunda es que aunque hay bindings para diferentes lenguajes, es imposible competir con la ubicuidad de los sistemas de ficheros. Mucho antes de que haya un binding de CORBA para las librerías de un sistema, cualquier lenguaje que se decida utilizar tendrá alguna forma de manipular un sistema de ficheros,

seguramente en red.

La arquitectura que proponemos, Oni, supera en portabilidad y capacidad para soportar capacidades heterogéneas a todos sus competidores, ya que, repetimos, prácticamente todos los sistemas y lenguajes tienen formas de acceder a sistemas de ficheros. Además es sencillo establecer pasarelas y proxies si fuese necesario.

6.2. Automatización de tareas y reificación de la interfaz

Uno de los requisitos de diseño es la posibilidad de automatizar tareas controlando la interfaz de forma programática. La existencia de una interfaz clara entre la interfaz de usuario y la aplicación permite que un programa pueda mover o controlar los widgets, inspeccionar sus contenidos y modificarlos. Sin embargo, el hecho de que esta interfaz de programación sea un sistema de ficheros, hace que se pueda llegar mucho más lejos. Se pueden utilizar herramientas heredadas de Unix, de Windows o de Mac-Os para manipular ficheros, como pueden ser un programa explorador de ficheros, un editor o la propia shell para manipular y ver el contenido de los elementos gráficos implicados.

A continuación, presentamos dos ejemplos de las posibilidades que se abren gracias a la utilización de sistemas de ficheros como interfaz para automatizar tareas.

El primero es la migración de interfaces mediante el uso de un sencillo script de shell. Gracias al diseño de la arquitectura, migrar una interfaz de usuario no es más que copiar un árbol de ficheros de un sitio a otro. Para ello no hace falta más que un programa que empaquete o serialice árboles de ficheros y uno capaz de desempaquetarlos. Este programa es **tar** y el script es el siguiente:

```
#!/bin/rc
{ cd /uifijo/; tar vent:win:* .} | {cd /uiportatil/; tar xvT}
```

El sistema de anuncios de Plan B se encargará de anunciar este nuevo árbol a la red y las aplicaciones interesadas, que habrán montado el anuncio, descubrirán la aparición de una nueva vista automáticamente. El script que realiza la copia, puede encontrarse en una máquina diferente tanto de las que sirven la interfaz como de los que lo utilizan.

Utilizando este mismo programa, se pueden hacer backups periódicos del contenido de la interfaz, que pueden tener diferentes usos, como por ejemplo darle contexto a otros programas.

El segundo programa del que hablaremos es un script para cerrar el sistema. Salva todo el estado que pueda haber por salvar y cierra el sistema. Para ello busca todos los botones de nombre **Save** y los aprieta.

```
#!/bin/rc

btsf= `du -a|grep '/ui/.*Save:but:[0-9]*$'`
if(! ~ $btsf '')
     for(b in $btsf){
          nameb = `{echo $b| sed 's/.*(Save:but:[0-9]*).*/\1/'}
          echo $nameb^' on' > $bf
     }
fshalt
echo halt > /dev/reboot
```

Este script no sólo se ha beneficiado de la reificación, que es indispensable, pues permite controlar de forma automática la interfaz e interactuar con ella de forma automática. Se ha beneficiado también de la convención de que todos los botones para salvar se llaman **Save**. Las convenciones como esta son indispensables si se quieren automatizar tareas.

De nuevo, muchos sistemas tienen un cierto nivel de reificación, desde Morphic [83] a los citados anteriormente, que utilizan CORBA y DCOM. Sin embargo, en ellos hay que utilizar aplicaciones especiales para imprimir o consultar el es-

tado de los objetos. Algunos incluso definen pasarelas a sistemas de ficheros que permiten realizar operaciones de consulta sobre ellos.

Oni lleva todas estas ideas un paso más allá. La existencia de una *única interfaz* de comunicación abierta con la interfaz de usuario, el sistema de ficheros, permite interponerse o la manipulación de la interfaz de usuario por comandos externos. Estos comandos no se tienen que escribir especialmente para Oni ni es necesario utilizar un lenguaje de programación. Los comandos estándar de manipulación de ficheros y los lenguajes de scripting de shell se pueden utilizar sin modificación para inspeccionar la interfaz de usuario y tratarlo automáticamente.

Aunque no es necesario, ni es objetivo nuestro lograr viveza ni llaneza, la reificación de la interfaz es lo suficientemente completa, para que una *implementación concreta* de Oni, la exponga a través de la interfaz de usuario, si se requieren estas propiedades.

6.3. Adaptabilidad

La adaptabilidad de Oni se basa principalmente en la separación entre la interfaz de usuario y la aplicación. Esta separación unida al uso de un alto nivel de abstracción en la comunicación permiten que se realice adaptación en dos niveles diferentes.

Por un lado, diferentes implementaciones de Oni, con diferentes modos de interacción y de comunicación con el usuario se pueden usar de forma transparente, ya sea como vistas ya sea como alternativas para comunicarse con la misma aplicación. Gracias a la interfaz de separación y el alto nivel de abstracción en la comunicación entre la aplicación y las diferentes implementaciones, se aísla lógica y presentación.

Por otro lado, esa misma interfaz lógica-presentación, al reificar la interfaz de usuario y ofrecer en general una interfaz abstracta y exponerla, permite automatizar el acceso a la interfaz o realizar tareas de adaptación sobre una misma

implementación de la interfaz de usuario; Ej: leer los nombres de los widgets del sistema de ficheros. Esta adaptación unida a automatización puede actuar también sobre diferentes vistas. La adaptación permite desde manipular las interfaces de usuario basándose en entradas multimodales a cambiar el tamaño de fuente dependiendo de la lejanía del usuario. Al fin y al cabo sólo se trata de monitorizar y manipular un sistema de ficheros.

6.4. Protección

Desde el momento en el que existe transparencia de red, hace falta protección. La protección tiene muchas facetas, en nuestro caso nos interesan principalmente dos, autenticación y confidencialidad.

La autenticación sirve para impedir que alguien que no tiene permiso para acceder a una interfaz de usuario o a una aplicación lo utilice. Permite también utilizar listas de control de acceso y permitir o rechazar el acceso a grupos de usuarios.

La confidencialidad impedirá que nadie sin permiso pueda leer los datos contenidos en la interfaz o en la aplicación o transmitidos entre ambos. La confidencialidad suele venir asociada a integridad que es asegurarse de que los datos transmitidos por un usuario con permisos lleguen al destinatario, en nuestro caso aplicación o interfaz tal y como fueran transmitidos.

Se han estudiado ampliamente varios mecanismos de protección para sistemas de ficheros en red. Por ejemplo AFS [38] utiliza Kerberos [132] para autenticación y cifrado, que provee confidencialidad. Muchos sistemas funcionan de forma similar, utilizando Kerberos o SSL [131]. Otros sistemas de ficheros integran el cifrado en su interior como CFS [33], permitiendo apilar diferentes niveles de cifrado.

Al final, en la mayor parte de estos sistemas, los usuarios se autentifican ante una autoridad central, lo que les permite acceder a un árbol de ficheros, con o sin cifrado. Una vez establecida la identidad y/o la pertenencia a un grupo, se

pueden establecer políticas de acceso a los recursos refinadas, como las ACL de Unix [26] y los permisos de ejecución lectura o escritura que forman parte de los metadatos de ficheros y directorios en Unix.

Todos estos mecanismos son conocidos y los usuarios están familiarizados con ellos lo que influye positivamente en la seguridad [123]. Además, al tratarse de mecanismos ampliamente estudiados y modelos probados, se reduce la posibilidad de que existan fallos de diseño. Todos estos mecanismos están basados en una autoridad central. En algunas circunstancias, cuando los usuarios se encuentren desconectados de una autoridad central, puede resultar útil utilizar los mecanismos de autenticación de Plan B suministrados por Shad [130], que permite autenticación personal entre pares sin depender de una autoridad central.

6.5. Mantenimiento transparente de vistas

La transparencia de localización se obtiene gracias a dos mecanismos principalmente. El primero es el anuncio de nuevos sistemas de ficheros, al que ya nos referimos al explicar la relación de pertenencia.

Una vez montado el sistema de ficheros en un lugar designado por convención y avisada de una forma o de otra la aplicación, ya no es necesaria ninguna referencia a la localización, que es completamente transparente. El mantenimiento transparente de vistas se puede obtener de forma similar mediante una librería o programa que sincronice de forma transparente N réplicas. Basta con que las operaciones realizadas se propaguen a todas las réplicas. La forma de hacer esto en un sistema de ficheros es bien conocida, por ejemplo, LOCUS [150].

El otro problema que existe con el mantenimiento transparente de vistas, es la transparencia. Como ya se mencionó en la introducción, en muchos casos, no se pueden utilizar interfaces remotos o vistas, debido al retardo o bajo ancho de banda de la red. Oni, igual que hacía Protium [158] baja el ancho de banda necesario, gracias a dos aspectos fundamentales:

- Eleva el nivel de abstracción de la comunicación entre la interfaz y la aplicación. Al no ocuparse de detalles de bajo nivel, hay un mayor contenido semántico en cada mensaje intercambiado entre la aplicación y la interfaz. En X Window System [121] o VNC [143] un mensaje puede decir "dibuja este pixel". En Oni dice, "Hay un nuevo botón", que equivale a miles de operaciones individuales de dibujar píxeles.

- El contenido local al interfaz de usuario se mantiene en él. Sólo la parte de la interacción y contenido que necesita la aplicación se transmite a ésta. Por ejemplo, las fuentes están siempre en local en la interfaz y nunca viajan por la red.

6.6. Recolección de basura y protección

Dos problemas nuevos que surgieron y que no planteamos en los requisitos fueron la recolección de basura y la protección. Aquí con protección nos referimos a protección en el sentido programático, no en el de seguridad. Tratamos ambos bajo el mismo epígrafe, porque hay una conexión sutil entre ambos, ya que una recolección de basura cuidadosa aumenta el nivel de protección.

6.6.1. Recolección de basura

La recolección de basura siempre es un problema en un entorno con recursos distribuidos. Aún más si existen vistas, es decir, hay varios interfaces para un mismo programa. Desde el punto de vista del sistema de ficheros de Oni, la recolección de basura en el caso normal no es complicada, ya que contamos con una conexión de los clientes. Si nada falla basta con detectar que se ha cerrado la conexión para borrar la interfaz de usuario. Este mecanismo sencillo además nos permite reproducir interfaces de usuario que hayamos archivado, por ejemplo para examinarlos sin la necesidad de tener clientes. Basta con abrir una conexión con el sistema de ficheros y crear y manipular los ficheros y directorios

adecuados.

El problema que se puede producir es que clientes se queden bloqueados en estados intermedios por un error de programación. En este caso la conexión se puede quedar abierta y no producirse la recolección de basura. Para ello hemos establecido un lease que se ocupa de cerrar la conexión y recolectar la interfaz si no hay actividad en el transcurso de un minuto. Un sencillo script que se encarga de hacer un stat de un fichero, mantiene la conexión abierta para el caso del interfaz extraído de un archivo.

Estas estrategias recolectan la basura desde el lado de la interfaz. El caso opuesto, es aquel en el que se precisa cerrar un programa si han desparecido todos sus interfaces. En este caso, también se tienen conexiones abiertas que se pueden comprobar. También hemos utilizado un lease para comprobar si existe actividad en las mismas.

En la figura 6.1 se puede ver la recolección de basura desde el punto de vista del sistema de ficheros de widgets definido por Oni (izquierda) y desde el punto de vista del cliente (derecha).

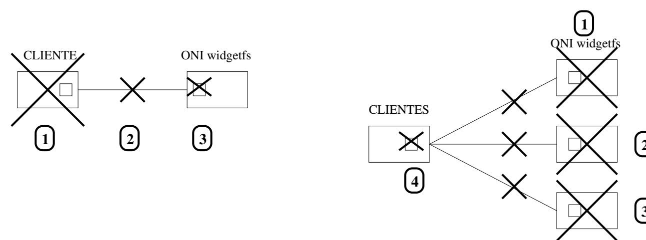

Figura 6.1: *Recolección de basura desde ambos puntos de vista*

Tanto en un sentido como en otro es necesario que se realicen operaciones cada cierto tiempo, como puede ser medio minuto o más que actúen como un latido para mantener abiertas las conexiones renovando el lease.

6.6.2. Protección

El otro problema del que hablábamos es la protección. En Oni la interfaz de usuario es un programa que se ocupa de atender a muchos clientes. Aunque hay un programa por ventana, hay datos compartidos, como el sistema de ficheros y el manejo de eventos. Si en cualquier momento una operación, que se suele deber a un cliente mal programado, bloquea la interfaz, perdemos el estado de varios programas y además, nos quedamos sin interfaz para la máquina. Si Oni es el único interfaz, esto puede suponer la necesidad de reiniciar la máquina.

Con esto en mente, se ha programado un prototipo de Oni teniendo especial cuidado en que un cliente que no se comporte bien, ya sea leyendo eventos o con un fichero de la interfaz, no pueda, de forma sencilla bloquear la interfaz. Es por ello que se han utilizado tantos procesos para programar y se ha realizado todo de forma cuidadosa. También se han añadido algunas heurísticas de recuento de eventos y widgets, de forma que si un programa parece haber perdido el control, no se permite que prosiga. A pesar de todo el cuidado que se haya podido tener, si un cliente se comporta de forma suficientemente inadecuada, puede bloquear la interfaz. En el momento que se tiene un servidor que provee a varios clientes, existe esta posibilidad. Existe un compromiso siempre entre protección y compartición de recursos y abstracción de los mismos. Este compromiso se establece a diferentes niveles en servidores como X Window System [121] o rio [79]. Creemos que subir el nivel de abstracción y compartir la interfaz de la forma en lo que lo hace Oni es lo suficientemente útil como para compensar por la pérdida de protección.

6.7. Rendimiento

No se han realizado medidas de rendimiento del sistema, puesto que no resulta necesario. En una interfaz de usuario, los tiempos de respuesta del sistema han de ser simplemente imperceptibles por el usuario.

Para ello implementamos el prototipo de Oni como prueba de concepto. El

prototipo fue escrito sin ninguna optimización y aún así no se percibe casi diferencias de rendimiento entre el uso de Oni y el de otra aplicación hecha con la misma librería de widgets, control. Esto es debido a que la mayor parte de las interacciones se encuentran contenidas en la interfaz de usuario, es decir las modificaciones que realiza el usuario se actualizan instantáneamente en la interfaz (cómo de rápido se actualizan depende en realidad de la librería de widgets con la que se escriba la implementación de Oni). Sólo en algunos eventos de actualización, como la creación de widgets o eventos que requieren la contestación de la aplicación para la actualización del interfaz y cruzan hasta el sistema de ficheros se percibe un retraso algo mayor, pero tolerable por el usuario. El prototipo de Oni se ha llegado a ejecutar en un Plan 9 sobre una IPAQ. Esta configuración se mostró en el WMCSA [92].

Adicionalmente la mayor parte del tiempo, los hilos de la interfaz de usuario del prototipo se encuentran bloqueados en pipes o canales, el prototipo consume muy pocos recursos de CPU. Sin embargo, esto depende fuertemente de la implementación de Oni. La prueba de concepto simplemente demuestra que se puede realizar una implementación de Oni que sea utilizable y sin consumir demasiados recursos.

Capítulo 7

Conclusiones y trabajo futuro

Se ha presentado una arquitectura novedosa para la construcción de interfaces de usuario. Esta arquitectura se basa en el uso de un sistema de ficheros representando a widgets y con un alto nivel de abstracción en las operaciones.

Como protocolo de comunicación entre la interfaz de usuario y los clientes se utiliza un sistema de ficheros. Esta idea, que ha demostrado ser particularmente poderosa, apareció por primera vez en Unix y luego se extendió en Plan 9 para exportar todos los recursos del sistema como ficheros. Al utilizar un sistema de ficheros como interfaz general de comunicación, se obtienen una serie de propiedades indispensables de forma automática. Estas propiedades son:

- Un sistema de nombrado que permite acceder de forma portable, uniforme y distribuida a los recursos.
- Mecanismos de protección probados y bien conocidos de protección, replicación y transparencia de red.
- Un sistema de anuncios y suscripciones, gracias al modelo de Plan B de anuncios y montajes.
- Un sistema distribuido de propagación de eventos.

Todas estas ventajas se obtienen por un precio mucho menor tanto en complejidad conceptual, como de programación de lo que cuestan en CORBA [149] o Jini [25].

Los objetos presentados por el servidor de interfaces de usuario a los clientes son widgets, elementos de alto nivel autocontenidos. Estos elementos son independientes y se relacionan entre sí mediante metaelementos representando relaciones de agrupación y espaciales. Gracias a la utilización de widgets autocontenidos expuestos a través de una interfaz que permite su manipulación a través de la red de forma portable, se logra un alto grado de reificación. Esto permite programar el sistema y automatizarlo de forma sencilla, en la mayor parte de los casos utilizando herramientas de propósito general que ya existían.

Tanto los elementos, como los mensajes que se intercambian a través del sistema de ficheros, describiendo eventos, manipulación del contenido y las relaciones entre widgets son un alto nivel de abstracción. Gracias a ello, es posible mantener la separación lógica-presentación, fundamental para permitir adaptación.

Como se ha demostrado, estas tres características permiten que se puedan superar los desafíos presentados por la visión de Weiser de hacer desaparecer los ordenadores de la atención del usuario. Estos desafíos eran consecuencia de la heterogeneidad tanto de los dispositivos como las redes y la modalidad de interacción con ellos además de la dinamicidad del entorno.

7.1. Limitaciones de la solución

La utilización de un sistema de ficheros como interfaz para exponer los widgets, tiene por supuesto algunas limitaciones. Estas limitaciones vienen dadas por la capacidad expresiva del sistema de ficheros. Existe un compromiso entre la riqueza de la interacción que se ofrece al usuario en la interfaz de usuario y la complejidad del sistema de ficheros que se ofrece al programador de aplica-

ciones. El caso más evidente que ilustra este compromiso es el número de tipos diferentes de tipos de widgets que se incluyen en el interfaz. Un mayor número y una mayor variedad de widgets pueden ofrecer una mayor riqueza de interacción con el usuario. Por otro lado comprometen la simplicidad de la implementación del sistema de ficheros, una componente crítica del sistema y además complican la programación para el usuario. En este compromiso, nos hemos decantado por simplificar el sistema de ficheros, en algunos casos en detrimento de la usabilidad. Este compromiso es la mayor limitación de la solución expuesta.

7.2. Contribuciones

Las contribuciones que aporta esta tesis al campo de la investigación en sistema ubicuos, son principalmente tres:

Identificación de la problemática asociada a interfaces de usuario en entornos ubicuos. La identificación de esta problemática, se ha traducido en los requisitos de diseño de la arquitectura Oni:

- Interoperabilidad entre dispositivos heterogéneos y adaptación a diferentes capacidades.
- Reflexión y programabilidad mediante la reificación, lo que permite adaptarse a la dinamicidad del entorno.
- Adaptación modal, separación de lógica y presentación para adaptar la interfaz a las necesidades del usuario y del entorno.
- Un modelo de protección adecuado a dicho entorno.
- Mantenimiento transparente de vistas y migración, lo que permite adaptar la interfaz a las necesidades del usuario.

Diseño de una arquitectura para estas interfaces. Diseño mediante la utilización de un sistema de ficheros de widgets especialmente adaptado a la

problemática de los sistemas ubicuos. Este diseño es la parte central de la presente tesis y ha sido analizado en cada una de sus decisiones de diseño.

Validación mediante la comprobación de los requisitos y la implementación de un prototipo que verifica la viabilidad de la aproximación tomada. Se ha implementado un prototipo de la arquitectura propuesta y se ha utilizado con el fin de comprobar que la aproximación era válida.

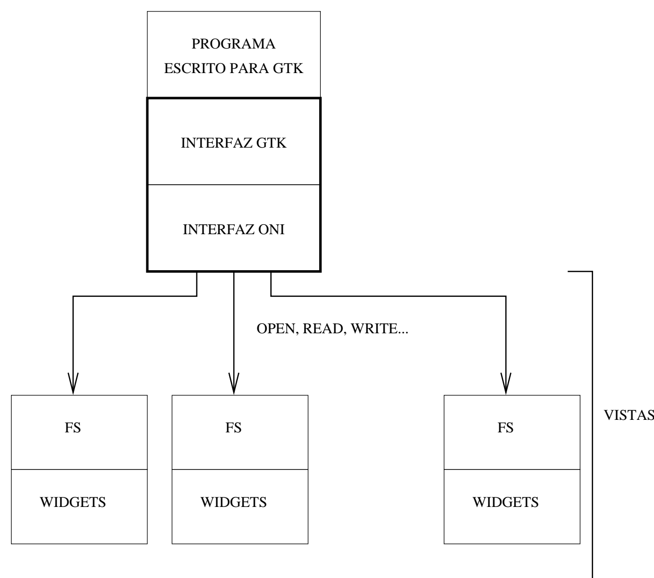

Figura 7.1: *Librería de adaptación para programas escritos para GTK*

7.3. Trabajo Futuro

Hay varias lineas de trabajo futuro que se pueden adoptar a partir del diseño de la arquitectura.

La primera, por supuesto, es la escritura de varias implementaciones multimodales de la interfaz y examinar la usabilidad de las aplicaciones cuando se utilizan diferentes vistas de las mismas.

La segunda y no menos importante, es explorar la adaptación de librerías que exporten hacia el programador una interfaz similar a AWT o GTK y que por

debajo utilicen nuestra infraestructura. Esto permitiría compilar aplicaciones heredadas y utilizarlas sin modificación. La arquitectura de este subsistema se puede ver en la figura 7.1.

Otra línea de trabajo interesante es la exploración de las diferentes formas que se pueden utilizar para automatizar el sistema basándose en parámetros como el contexto. La reificación de la interfaz permite, como ya hemos visto, que un programa o script consulte en cada instante el estado de la interfaz de usuario. Gracias a esto se tiene nueva información de contexto proveniente de la interfaz. Esta nueva información se puede combinar con la información que ya existe y utilizar la reificación de la interfaz para realizar tareas de forma automática. Decidir cómo se debe combinar esta información, es una tarea compleja que será el objeto de futuras investigaciones. Los mecanismos para realizarla se encuentran ya presentes en Oni.

Bibliografía

[1] *Java AWT Reference.* O'Reilly, 1997.

[2] *Java Swing.* O'Reilly Sebastopol, Calif, 1998.

[3] SUPPLE: automatically generating user interfaces. *Proceedings of the 9th international conference on Intelligent user interface*, 2001.

[4] Exposé. *http://www.apple.com/macosx/features/expose/*, 2003.

[5] Players and costumes. *http://tweak.impara.de/TECHNOLOGY/Whitepapers/PlayersandCostumes*, 2003.

[6] A Unifying Reference Framework for the Development of Plastic User Interfaces. *Proceedings of the 9th international conference on Intelligent user interface*, pages 93–100, 2004.

[7] Short introduction into tweak. *http://tweak.impara.de/TECHNOLOGY/Tutorials*, 2004.

[8] XHTML+Voice Profile 1.1. *http://www-306.ibm.com/software/pervasive/multimodal/x+v/11/spec.htm*, 2004.

[9] Command line interfaces. *http://en.wikipedia.org/wiki/Command_ line_ interface*, 2005.

[10] *Cross-Platform GUI Programming with wxWidgets (Bruce Perens Open Source)*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2005.

[11] DsY-Windows. *http://sourceforge.net/projects/dsywindows*, 2005.

[12] Google Maps. *http://maps.google.com*, 2005.

[13] interfaz. *http://es.wikipedia.org/wiki/Interfaz*, 2005.

[14] Aquarela. *http://www.zen34585.zen.co.uk/dist/aquarela/index.html*, 2006.

[15] NeXT. *http://en.wikipedia.org/wiki/NeXTStep*, 2006.

[16] Plan 9 Remote Resource Protocol. *http://v9fs.sourceforge.net/rfc/*, 2006.

[17] Quartz. *http://developer.apple.com/graphicsimaging/quartz/*, 2006.

[18] Quartz Compositor. *http://en.wikipedia.org/wiki/Quartz_ Compositor*, 2006.

[19] Beryl. *http://www.beryl-project.org*, 2007.

[20] E. Aarts et al. *Ambient intelligence*. Springer-Verlag, 2005.

[21] A. Alireza, U. Lang, M. Padelis, R. Schreiner, and M. Schumacher. The challenges of corba security.
http://citeseer.ist.psu.edu/393276.html, 2000.

[22] B. M. Andy Bower. Twisting the triad. *Tutorial Paper for ESUG 2000*, 2000.

[23] Apple. Cocoa. *http://developer.apple.com/cocoa*, 2005.

[24] K. Arnold and J. Gosling. *The Java programming language.* ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 1998.

[25] K. Arnold, R. Scheifler, J. Waldo, B. O'Sullivan, A. Wollrath, B. O'Sullivan, and A. Wollrath. *Jini Specification.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[26] M. Bach. *The design of the UNIX operating system.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1986.

[27] F. Ballesteros, K. Leal, G. Guardiola, and E. Soriano. The Design and Implementation of Plan B 3rd edition. A dynamic distributed computing environment. *Percom*, 2006.

[28] S. Beale. Mac OS X: Imaging in Quartz (and 3-D). *MacWEEK http://macweek.zdnet. com/2000/08/20/0824moveeight.html*, 24, Aug. 2000.

[29] B. Bederson and B. McAlister. *Jazz: An Extensible 2D+ zooming Graphics Toolkit in Java.* Human-Computer Interaction Laboratory, Center for Automation Research, 1999.

[30] T. Berners-Lee and D. Connolly. RFC 1866: Hypertext Markup Language -2.0. *Status: PROPOSED STANDARD*, Aug. 1995.

[31] T. Berners-Lee, R. Fielding, and L. Masinter. RFC2396: Uniform Resource Identifiers (URI): Generic Syntax. *Internet RFCs*, 1998.

[32] T. Bienz and R. Cohn. *Portable document format reference manual.* Addison-Wesley, 1993.

[33] M. Blaze. A cryptographic file system for UNIX. *Proceedings of the 1st ACM conference on Computer and communications security*, pages 9–16, 1993.

[34] D. Bovet and M. Cesati. *Understanding the Linux Kernel.* O'Reilly Media, 2003.

[35] K. Brockschmidt. *Inside OLE.* Microsoft Press Redmond, WA, USA, 1995.

[36] T. Browne. Using declarative descriptions to model user interfaces with mastermind.
http://citeseer.ist.psu.edu/browne97using.html, 1997.

[37] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. Technical report, RFC 1813, Network Working Group, June 1995.

[38] R. Campbell. *Managing AFS: The Andrew File System.* Prentice Hall, 1998.

[39] M. Chapman. Create user interfaces with Glade. *Linux Journal*, 87(88):90–92.

[40] H. Chen, T. Finin, A. Joshi, L. Kagal, F. Perich, and D. Chakraborty. Intelligent agents meet the semantic Web in smart spaces. *IEEE Internet Computing*, 8(6):69–79, 2004.

[41] I. G. Chris Sells. *Programming Windows presentation foundation.* O'Reilly, 2005.

[42] L. de sistemas. Laboratorio de sistemas. *http://lsub.org*, 2006.

[43] D. J. Dionne and M. Durrant. Uclinux homepage. *http://www.uclinux.org/*, 2002.

[44] A. Dix, J. Finley, G. Abowd, and R. Beale. *Human-computer interaction.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1998.

[45] K. Ducatel et al. Scenarios for Ambient Intelligence in 2010, 2001.

[46] D. Faure. The kde project. *http://developer.kde.org/documentation/tutorials/kparts*, 2006.

[47] J. Feiler. *Mac OS X developer's guide.* Morgan Kaufmann, San Diego, 2002.

[48] J. Fisher. Securing x windows. 1995.

[49] J. Fisher. Securing x windows, ucrl-ma-121788, ciac-2316 r.0. 1995.

[50] D. Flanagan. *JavaScript: the definitive guide*. O'Reilly, 1997.

[51] K. L. A. E. S. P. d. l. H. Q. E. M. C. A. L. Francisco J. Ballesteros, Gorka Guardiola Muzquiz and S. Arévalo. Plan B: Boxes for network resources. *Journal of Brasilian Computer Society Special issue on Adaptable Computing Systems, ISSN 0104-6500*, 2004.

[52] J. Friedl. *Mastering Regular Expressions*. O'Reilly, 2002.

[53] W. Furmanski. Interface to NeWS in MOVIE. Technical report, NPAC Technical Report NPAC/MOVIE/92–10.

[54] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[55] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project Aura: toward distraction-free pervasive computing. *Pervasive Computing, IEEE*, 1(2):22–31, 2002.

[56] N. Gershenfeld. *When Things Start to Think*. Owl Books, 2000.

[57] J. Gettys and R. Scheifler. *Xlib: C Language X interface*. The X Consortium Inc., 1996.

[58] E. Giguère. *Java 2 micro edition*. Wiley Computer Pub, 2000.

[59] I. GmbH. Islands whitepapers. *http://tweak.impara.de/TECHNOLOGY/Whitepapers/Islands/*.

[60] A. Goldberg and D. Robson. Smalltalk-80: The language and its implementation. 1983.

[61] J. Gosling, D. Rosenthal, and M. Arden. *The NeWS book: an introduction to the network/extensible window system*. Springer-Verlag New York, Inc. New York, NY, USA, 1989.

[62] K. Granroth. Using KDE components (KParts), 2000. *Unpublished invited talk at Annual Linux Showcase*, 2000.

[63] R. Grimes and R. Grimes. *Professional Dcom Programming*. Wrox Press Ltd. Birmingham, UK, UK, 1997.

[64] P. Haahr. Montage: Breaking windows into small pieces. *http://www.webcom.com/haahr/montage/montage-usenix-s90.html*.

[65] J. Harper. Sawmill: an extensible window manager, 2005.

[66] A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C# Programming Language*. Addison Wesley, 2006.

[67] S. Hjelm. Visualizing the Vague: Invisible Computers in Contemporary Design. *Design Issues*, 21(2):72, 2005.

[68] C. Hoare. A theory of CSP. *Commun. ACM*, 21(8), 1978.

[69] IBM. Systems application architecture: Basic interface design guide, document sc26-4583-0. 1990.

[70] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. *ACM SIGPLAN Notices*, 32(10):318–326, 1997.

[71] K. P. James Gettys. The (re)architecture of the x window system. 2004.

[72] P. R. J.M. Arfman. Project athena: Supporting distributed computer at mit. *IBM Systems Journal*, 31(3), 1992.

[73] B. Johnson, M. Young, and C. Skibo. *Inside Microsoft Visual Studio .NET*. Microsoft Press Redmond, WA, USA, 2002.

[74] B. Kernighan and R. Pike. *The UNIX Programming Environment*. Prentice Hall Professional Technical Reference, 1984.

[75] B. Kernighan and D. Ritchie. *The C programming language*. Prentice-Hall, 1988.

[76] T. Killian. Processes as Files. *USENIX Summer Conference Proceedings*.

[77] G. Krasner and S. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object Oriented Programming*, August/September 1998.

[78] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, pages 26–49, Aug. 1988.

[79] B. Labs. *Plan 9 operating system manual [Manual pages]*. Vitanuova, 2002.

[80] L. Lamb and A. Robbins. *Learning the Vi Editor*. O'Reilly, 1998.

[81] W. Leler. PIX, the latest NeWS. *COMPCON Spring'89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage, Digest of Papers.*, pages 239–242.

[82] C. Locke. Styx on a brick. *http://www.vitanuova.com/inferno/rcx_paper.html*, 2006.

[83] J. I. Maloney and R. B. Smith. Directness and liveness in the morphic user interface construction environment. *Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 21–28, 1995.

[84] T. Mansfield and R. Taylor. The Orbit Collaboration Framework. 1997.

[85] E. L. Mark Thomas, D. Rückert. The y window system. *http://www.y-windows.org*, 2006.

[86] A. C. Martín. *Proyecto de fin de carrera: Ingeniería de telecomunicación, Procesado de eventos de raton para el sistema operativo Plan 9*. Universidad Carlos III, 2004.

[87] J. Mayes. Multimedia: a perspective. *Multimedia: the Future of User Interfaces, IEE Colloquium on*, page 1, 1990.

[88] T. Merz. *PostScript & Acrobat/PDF: Applications, Troubleshooting, and Cross-platform Publishing*. Springer, 1997.

[89] Microsoft. Com: Component object model technologies. *http://www.microsoft.com/com*, 2006.

[90] Microsoft. Get started using remote desktop with windows xp professional. *http://www.microsoft.com/windowsxp/using/mobility/getstarted/remoteintro.mspx*, 2006.

[91] M. Mueller-Prove. Vision and reality of user centered interfaces: Apple lisa. *http://www.mprove.de/diplom/text/3.1.8_ lisa.html*, 2004.

[92] G. G. Muzquiz and K. L. Algara. A Mobile and ubiquitous system for the new millennium. Doing new things with ancient technology from the 70s!. 2004.

[93] OMG. Iinterface definition language specification standard. 2005.

[94] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[95] K. Packard. Design and implementation of the x rendering extension. June 2001.

[96] R. Pike. Acme: A User Interface for Programmers. *Proc. USENIX 1994 Winter Technical Conference.*

[97] R. Pike. The blit: A multiplexed graphics terminal. *Bell Labs Tech. J.*, 63(8, part 2):1607–1631, 1984.

[98] R. Pike. Structural Regular Expressions. 1987.

[99] R. Pike. The Text Editor sam. *Software - Practice and Experience*, 17(11):813–845, 1987.

[100] R. Pike. Window systems should be transparent. *Computing Systems*, 1(3):279–296, Summer 1988.

[101] R. Pike. $8^{1/2}$, the Plan 9 window system. In *Proceedings of the Summer 1991 USENIX Conference, Nashville, TN, USA, June 10–14, 1991*, pages 257–265 (of x 473), Berkeley, CA, USA, 1991. USENIX.

[102] R. Pike, B. Locanthi, and J. Reiser. Hardware/Software Trade-offs for Bitmap Graphics on the Blit. *Software - Practice and Experience*, 15(2):131–151, 1985.

[103] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from bell labs. In *UKUUG, Proceedings of the Summer 1990 Conference*, London, England, July 1990.

[104] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The Use of Name Spaces in Plan 9.

[105] R. Pike and D. Ritchie. The Styx® architecture for distributed systems. *Bell Labs Technical Journal*, 4(2):146–152, 1999.

[106] L. Pinson and R. Wiener. *Objective-C: object-oriented programming techniques*. Addison-Wesley, 1991.

[107] S. Ponnekanti, B. Johanson, E. Kiciman, and A. Fox. Portability, extensibility and robustness in iROS. pages 11–19, 2003.

[108] T. Porter and T. Duff. Compositing digital images. *Computer Graphics*, 18(3):253–259, July 1984.

[109] T. F. Project. Fresco homepage. *http://www.fresco.org*, 2004.

[110] T. G. Project. Bonobo document model. *http://developer.gnome.org/arch/gnome/componentmodel/bonobo.html*, 2006.

[111] T. G. Project. The gnome project. *http://gnome.org*, 2006.

[112] T. K. Project. The kde project. *http://kde.org*, 2006.

[113] A. Puder. XML11–An Abstract Windowing Protocol. *Science of Computer Programming*, 59(1-2):97–108, 2006.

[114] T. C. Qnx. Ross a. mckegney student : 426 7519 7 june 2000 rmc eee 551 – qnx realtime operating systems 1. about qssl. *http://citeseer.ist.psu.edu/538424.html*.

[115] T. C. Qnx. Qnx neutrino os developer support, 2005.

[116] M. Reiser and N. Wirth. Project oberon - the design of an operating system and compiler. 1992.

[117] R. Riggs. *Programming wireless devices with the Java 2 platform, micro edition*. Addison-Wesley Professional, 2003.

[118] M. Russinovich and D. Solomon. *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*. Microsoft Press, 2005.

[119] M. Satyanarayanan. Pervasive computing: vision and challenges. *Personal Communications, IEEE [see also IEEE Wireless Communications]*, 8(4):10–17, 2001.

[120] R. Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, 1997.

[121] R. W. Scheifler and J. Gettys. *X Window System*. Digital Press, 1992.

[122] B. Schilit and U. Sengupta. Device ensembles [ubiquitous computing]. *Computer*, 37(12):56–64, 2004.

[123] B. Schneier. *Beyond Fear: thinking sensibly about security in an uncertain world.* Copernicus Books, New York, NY, 2003.

[124] S. Schubiger-Banz, S. Maffioletti, B. Hirsbrunner, and A. Tafat-Bouzid. Providing service in a changing ubiquitous computing environment. *Proc. of the Workshop on Infrastructure for Smart Devices-How to Make Ubiquity an Actuality (HUC 2000), September*, 2000.

[125] R. Schulmeister. *Grundlagen hypermedialer Lernsysteme: Theorie-Didaktik-Design.* Oldenbourg, 2002.

[126] M. M. SL Garfinkel. *NeXTSTEP programming.* Springer, 1993.

[127] A. Smailagic, D. Siewiorek, L. Bass, B. Iannucci, A. Dahbura, S. Eddleston, B. Hanson, and E. Chang. MoCCA: a Mobile Communication and Computing Architecture. *Wearable Computers, 1999. Digest of Papers. The Third International Symposium on*, pages 64–71, 1999.

[128] D. Smith, A. Kay, A. Raab, and D. Reed. Croquet–A Collaboration System Architecture. *Proceedings of the First Conference on Creating, Connecting and Collaborating through Computing (C*, 5, 2003.

[129] P. D. Smith. Lbx mini-howto. *http://www.tldp.org/HOWTO/LBX.html*, 1997.

[130] E. Soriano. Shad: A human centered security architecture for partitionable, dynamic and heterogeneous distributed systems. In *Proceedings of the 5th ACM/IFIP/USENIX International Middleware Workshops (1st International Middleware Doctoral Symposium).* ACM Press, 2004.

[131] W. Stallings. SSL: Foundation for Web Security. *The Internet Protocol Journal*, 1(1):20–29, 1998.

[132] J. Steiner, C. Neuman, and J. Schiller. Kerberos: An Authentication Service for Open Network Systems. *Proc. Winter USENIX Conference*, 1988.

[133] W. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley, 1994.

[134] B. Stroustrup. *The C++ programming language.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1986.

[135] D. Sweet et al. *KDE 2.0 Development.* Sams, 2001.

[136] O. Symbian. Symbian OS: The Mobile Operating System.

[137] A. Tanenbaum and R. Van Renesse. Distributed operating systems. *ACM Computing Surveys (CSUR)*, 17(4):72, 1985.

[138] A. Tanenbaum and M. Van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR Upper Saddle River, NJ, USA, 2001.

[139] T. G. team. Gtk+ the gimp toolkit, 2005.

[140] T. Thai and H. Lam. *. Net Framework Essentials.* O'Reilly, 2003.

[141] S. Thomke and A. Nimgade. *BMW AG: The Digital Car Project.* Harvard Business School Pub. Corp, 1998.

[142] T. Thompson. The next step. *Byte*, 1989.

[143] K. R. W. Tristan Richardson. The rfb protocol. *http://www.cl.cam.ac.uk/Research/DTG/attarchive/vnc/rfbproto.pdf*, Aug. 1998.

[144] Trolltech. Trolltech webpage, 2006.

[145] A. van Dam. Post-wimp user interfaces. *Communications of the ACM*, 40:63–67, 1997.

[146] G. Van Rossum et al. Python programming language. *CWI, Department CST, The Netherlands*, 1994.

[147] R. Vertegaal. Designing attentive interfaces. *Proceedings of the symposium on Eye tracking research & applications*, pages 23–30, 2002.

[148] K. Vigor. Dxpc homepage. *http://www.vigor.nu/dxpc/*, 2006.

[149] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), Feb. 1997.

[150] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The locus distributed operating system. In *SOSP '83: Proceedings of the ninth ACM symposium on Operating systems principles*, pages 49–70, New York, NY, USA, 1983. ACM Press.

[151] R. Want, B. Schilit, N. Adams, R. Gold, K. Petersen, D. Goldberg, J. Ellis, and M. Weiser. An overview of the PARCTAB ubiquitous computing experiment.

[152] M. Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, 1999.

[153] M. Weiser. Some computer science issues in ubiquitous computing. *ACM SIGMOBILE Mobile Computing and Communications Review*, 3(3), 1999.

[154] M. Weiser and J. Brown. Designing Calm Technology. *PowerGrid Journal*, 1(1):75–85, 1996.

[155] M. Weiser and J. Brown. The coming age of calm technology [1], 1996.

[156] Wikipedia. Apple lisa. *http://en.wikipedia.org/wiki/Apple_Lisa*, 2004.

[157] S. C. y Mike Potel. *Inside Taligent Technology.* Addison Wesley, 1995.

[158] C. Young, L. YN, T. Szymanski, J. Reppy, R. Pike, G. Narlikar, S. Mullender, and E. Grosse. Protium, an infrastructure for partitioned applications. In *Proceedings of the Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS)*, pages 41–46, Schloss Elmau, Germany, 2001.

Índice alfabético

$8\frac{1}{2}$, 84

ACL, 159
acme, 85
Aleph, 83
API, 50
Apple, 70
ATK, 63
autenticación, 37, 91, 92, 113, 142, 158, 159
AWT, 64, 80, 168

backups, 156
Bart, 67
Bell, 67, 82, 83, 92
bitmaps, 88
Blit, 67, 68
Bob, 89
Bonobo, 32
boxes, 118

callbacks, 59, 62, 63
Caltech, 89
check, 118
Chomsky, 147
Cocoa, 70
Computer, 27, 86, 89
CORBA, 37, 65, 66, 92, 154, 156
costume, 74
CUA, 59

DCOM, 65, 66
DEC, 89
dinamicidad, 28, 37, 95, 166, 167
display, 111
Dodge, 92
DPS, 86–88

ed, 84
encapsulación, 53
Environment, 89
EOF, 140, 143
estructura, 139, 144

fid, 133
Forms, 64
framebuffer, 94
FrameMaker, 89
framework, 51, 59, 70, 72, 79
frameworks, 47, 51, 52, 58, 68, 85

Fresco, 92

gelem, 144
Gettys, 89
GIF, 119
GIMP, 62
GLib, 63
Google, 49
Gordon, 92
Gosling, 88
gproxies, 139
gproxy, 144
GTK, 129, 168

IBM, 58
interacción, 44
Interactive, 89
interoperabilidad, 74
Intuitive, 71
inutilizable, 91

JavaScript, 79
javascript, 79–81
Jim, 89
John, 61
JPG, 118, 119

KDE, 63
Kimball, 62
KPARTS, 67

Labs, 67, 82
lease, 161
Lib, 91
listener, 139
ls, 137

Mac, 49, 52, 87
mantenimiento transparente de vistas, 6, 102, 114, 159
Massachusetts, 89
Mattis, 62
medio, 44, 68, 107, 114, 161
metadatos, 159
metaelemento, 107
metalemento, 107
metalementos, 106, 114
Microsoft, 59
minilenguaje, 144, 145
MIT, 34
MMU, 29
modalidad, 44, 45, 166
modo, 35, 44, 45, 49, 110, 116, 119
morph, 73
Morphic, 73, 74
morphs, 73
Motif, 58–60, 89
Mouse, 139
multicast, 113
multimodales, 158, 168
Multitasking, 89
Mux, 82
MVC, 65, 68–72, 74

Network, 94

NeWS, 88, 89
NeXT, 86, 87
NeXTStep, 87
NFS, 154

Object, 71
Oni, 44, 59, 61, 62, 74, 76–78, 84, 97, 101, 110, 111, 135, 153, 154, 157, 160–162
Open, 58, 89
Openlook, 89
OSF, 58
Ousterhout, 61

píxeles, 23, 84, 89
Pango, 63
PDFs, 88
Peter, 62
Photon, 92
Pike, 67, 83
Plastic, 76
player, 74
Players, 74
Pointing, 47
portapapeles, 94
PostScript, 86, 87
postscript, 75
proc, 139
prompt, 45, 46
protección, 28, 34, 35, 38, 43, 58, 66, 74, 85, 95, 153, 158, 160, 162, 165, 167
Protium, 78, 79
proxies, 65, 112, 139, 155
proxy, 30, 138
pswraps, 87
Python, 62, 80

QID, 133
Qnx, 92, 93
Quartz, 71, 85–88
QuickDraw, 88

RAM, 67
recolección de basura, 160, 161
redimensionado, 60, 91
redimensionar, 82
reificación, 31, 35, 38, 74, 86, 112, 114, 115, 155
reificar, 66, 86, 157
reificarlo, 50
reinterpretación, 84, 94
relocalización, 1
render, 86
renderizado, 63
rio, 35, 83, 139
RPC, 70, 132, 133
RPCs, 30, 66, 131–133, 137–140, 143
runtimes, 153

sam, 85
Scgui, 80
Scheifler, 89
Scheme, 84

script, 155, 156, 161
scripting, 157
scripts, 62
SGI, 89
shell, 107, 155, 157
sistema de ventanas, 31, 41, 42, 46–48, 82–88, 90, 92
Smalltalk, 71, 74
Spencer, 62
stat, 161
stubs, 65, 75
Sun, 72, 88, 89
SUPPLE, 76
svg, 121
Systems, 71

Technology, 89
timer, 146
tokenización, 146, 147
tokenizan, 145
toolkit, 37, 87, 102, 131, 135
toolkits, 50, 58, 64, 85, 117
Tweak, 74, 75
Type, 140
typescript, 46

uifs, 139
usabilidad, 27, 44, 168

V1, 145, 147, 149
V2, 149–151

WeirdX, 80, 81
Weiser, 166
widget, 59, 72, 103–106, 108, 110, 112, 113, 116, 119–121, 123, 135, 138, 144
widgets, 6, 31, 50, 59, 60, 64, 65, 72, 75, 91, 96, 97, 103, 106, 108, 111, 112, 116, 117, 122, 125, 129, 135, 137, 139, 143, 165, 166
window, 89–91
Windows, 35, 59, 64, 66, 155
winkiller, 139
wxWidgets, 59, 60

XML, 75, 80