



Tesis de Ingeniería en Informática

Abstracción en el desarrollo de software independiente de la plataforma

Análisis del proceso de desarrollo de *Cross-Platform Support
Middlewares*

Autor: Patricio Zavolinsky (81.611)

(pzavolinsky@yahoo.com.ar)

Tutora: Lic. Adriana Echeverría

Índice

Índice	1
Introducción	4
1. Herramientas de análisis de un CPSM	6
1.1. Modelo formal	6
1.2. Parámetros característicos de un CPSM	11
1.2.1. Flexibilidad	11
1.2.2. Compatibilidad	14
1.2.3. Mantenimiento	16
1.2.4. Seguridad	19
1.2.5. Performance	21
1.2.6. Relación entre los parámetros característicos de un CPSM	21
1.3. Resumen	23
2. Desarrollo de un CPSM	25
2.1. Características generales de un CPSM	25
2.1.1. Selección de paradigma	25
2.1.2. Selección del lenguaje de programación	28
2.1.3. Propósito del CPSM	30
2.1.3.1. Flexibilidad vs. Seguridad	31
2.2. Definición de servicios provistos por el CPSM	34
2.2.1. Especificación de las interfaces de cada servicio provisto por el CPSM	35
2.3. Definición de plataformas compatibles con el CPSM	40
2.4. Mecanismos de abstracción en un CPSM	41
2.4.1. Implementaciones alternativas vs. implementaciones incompatibles	41
2.4.2. Compilación selectiva	41
2.4.2.1. Compilación condicional	42
2.4.2.2. Separación física	43
2.4.3. Selección de implementaciones alternativas de un servicio	50
2.4.4. Inicialización y Finalización de servicios	52
2.5. Resumen	57
3. Caso de Estudio	59
3.1. Descripción	59
3.2. Definición del CPSM	59
3.2.1. Especificación de las interfaces de los servicios provistos	61
3.2.1.1. Comunicaciones	61
3.2.1.2. Concurrencia	64
3.2.1.3. Bibliotecas dinámicas	68
3.2.1.4. Servicios comunes	70
3.2.2. Mecanismo de abstracción	72
3.3. Implementación del CPSM	76
3.3.1. Comunicaciones	77

3.3.2.	Concurrencia	86
3.3.3.	Bibliotecas dinámicas	93
3.3.4.	Servicios comunes	95
3.4.	Desarrollo de un programa sustentado sobre el CPSM	96
3.4.1.	Primer intento: Comunicaciones	96
3.4.2.	Segundo intento: Comunicaciones y Concurrencia	99
3.4.3.	Tercer intento: Comunicaciones, Concurrencia y Bibliotecas dinámicas	103
4.	Análisis y Comparación de CPSMs	121
4.1.	Netscape Portable Runtime (NSPR)	121
4.2.	ADAPTIVE Communication Environment (ACE)	125
4.3.	Simple DirectMedia Layer (SDL)	127
4.4.	Boost	129
4.5.	wxWidgets	130
4.6.	Resumen	131
5.	Conclusiones y futuras líneas de estudio	133
	Apéndice	136
A.	Glosario	136
	Referencias	138

Resumen

Un programa, en relación a la plataforma que lo sustenta, debe enfrentar un desafío: mantener su compatibilidad en el tiempo y el espacio. Es decir, permanecer compatible con la plataforma que lo sustenta a pesar de la evolución de ésta (compatibilidad en el tiempo), y ser compatible con la mayor cantidad de plataformas posible (compatibilidad en el espacio).

La solución tradicional a este problema consiste en concentrar los detalles propios de la plataforma en una capa de abstracción. El objetivo de esta capa es encapsular los detalles de las interfaces de programación provistas por distintas plataformas, en una única interfaz homogénea. Una capa de abstracción con estas características se denomina *Middleware*.

Un *Cross Platform Support Middleware* (CPSM) es un *Middleware* que garantiza que la interfaz que provee se encuentra implementada en su totalidad en todas las plataformas con las que es compatible.

El objetivo de esta tesis consistió en analizar las actividades involucradas, y los problemas que frecuentemente se deben enfrentar, en el desarrollo de un CPSM.

Dentro de las conclusiones derivadas del análisis presentado cabe destacar la relevancia del proceso de desarrollo de un CPSM y la reconstrucción de dicho proceso a partir de ejemplos completos de CPSMs que constituyen evidencia empírica de la viabilidad de construcción de los mismos.

Para el problema de inicialización y finalización de servicios se realizó un análisis de diferentes alternativas: inicialización y finalización explícitas e implícitas. Para el caso particular de inicialización implícita se propuso una solución original que, bajo ciertas restricciones, resuelve el problema a través de la creación una instancia estática de una clase *Initializer*.

Para ilustrar el desarrollo de un CPSM, se implementó un CPSM completo, como caso de estudio, que provee servicios de concurrencia, comunicaciones y asociación explícita de bibliotecas dinámicas en Microsoft Windows y GNU/Linux. Adicionalmente se construyó una aplicación sustentada sobre dicho CPSM que permite abstraer la interacción propia de una aplicación cliente/servidor (i.e. el protocolo de comunicaciones) de el establecimiento de la conexión (sobre TCP/IP). En conjunto, dicha aplicación y el CPSM del caso de estudio ejemplifican el proceso de desarrollo y la posterior utilización de un CPSM.

Introducción

El 14 de Octubre de 1987, Henry Spencer publicó sus “Diez Mandamientos para los Programadores C” en el grupo de noticias *comp.lang.c*[Spe87]. El décimo mandamiento ordenaba:

Thou shalt foreswear, renounce, and abjure the vile heresy which claimeth that “All the world’s a VAX”, and have no commerce with the benighted heathens who cling to this barbarous belief, that the days of thy program may be long even though the days of thy current machine be short.¹

Poco más de tres años más tarde, Spencer actualizó sus Mandamientos. En la versión Anotada de 1990[Spe90], agregó la siguiente nota al décimo mandamiento:

This particular heresy bids fair to be replaced by “All the world’s a Sun” or “All the world’s a 386” (this latter being a particularly revolting invention of Satan), but the words apply to all such without limitation. Beware, in particular, of the subtle and terrible “All the world’s a 32-bit machine”, which is almost true today but shall cease to be so before thy resume grows too much longer.²

Al margen del tono jocoso en el cual está formulado, el décimo mandamiento denuncia una preocupación que hoy en día no ha perdido vigencia. Los programas se sustentan sobre sistemas operativos, los sistemas operativos sobre arquitecturas de *hardware*. Todo aquello sobre lo que se construye un programa es considerado parte de la “plataforma”. Un programa, en relación a la plataforma que lo sustenta, debe enfrentar un desafío: mantener su compatibilidad en el tiempo y el espacio. Es decir, permanecer compatible con la plataforma que lo sustenta a pesar de la evolución de ésta (compatibilidad en el tiempo), y ser compatible con la mayor cantidad de plataformas posible (compatibilidad en el espacio).

El primer aspecto del desafío consiste en sobrevivir a las modificaciones en la plataforma subyacente, puntualmente, a modificaciones en las interfaces de programación (APIs) de la plataforma, como consecuencia de la evolución. El segundo, consiste en poseer compatibilidad con más de una plataforma. Una motivación para buscar este tipo de compatibilidad es aumentar los usuarios potenciales del programa, a todos los usuarios de las plataformas con las que el programa es compatible. En sistemas distribuidos, una motivación adicional es mejorar la tolerancia a fallos (i.e. *fault tolerance*), utilizando redundancia heterogénea, ya que utilizar programas redundantes sobre diferentes plataformas, permite adquirir cierta tolerancia a fallos en la plataforma³[VR01].

¹“Rechazarás, abandonarás y repudiarás la vil herejía que asegura que ‘Todo el mundo es una VAX’ y evitarás toda relación con los ignorantes incivilizados que se aferran a esta creencia barbárica, para que los días de tu programa sean largos incluso cuando los días de tu actual máquina sean cortos.”

²“Esta herejía, en particular, podría reemplazarse por ‘Todo el mundo es una Sun’ o ‘Todo el mundo es una 386’ (siendo esta última una particularmente desagradable invención de Satán), pero se aplica a todos los casos similares, sin excepción. Ten especial cuidado con la sutil y terrible ‘Todo el mundo es una máquina de 32 bits’, que es prácticamente cierta hoy en día, pero dejará de serlo antes de que tu currículum crezca demasiado.”

³Por ejemplo, un mismo programa que se ejecuta en varios *hosts*, algunos de ellos con la plataforma p^1 y otros con la plataforma p^2 . Supóngase la existencia de un *bug* en determinada secuencia de ejecución en p^1 , que no se manifiesta en p^2 . En este escenario, si el programa falla, como consecuencia del *bug* en p^1 , los *hosts* que utilizan p^2 no se verán afectados y el programa distribuido se degradará pero no dejará de funcionar.

La solución tradicional a este problema consiste en concentrar los detalles propios de la plataforma en una capa de abstracción⁴. El objetivo de esta capa es encapsular los detalles de las APIs provistas por distintas plataformas, en una única interfaz homogénea. Una capa de abstracción con estas características se denomina *Middleware*⁵[Tan02].

Un *Middleware* que garantiza que la interfaz que provee⁶ se encuentra implementada en su totalidad en todas las plataformas con las que es compatible será denominado *Cross Platform Support Middleware* (CPSM).

El objetivo de esta tesis es analizar las actividades involucradas, y los problemas que frecuentemente se deben enfrentar, en el desarrollo de un CPSM.

Este análisis se organiza de la siguiente manera:

- En el Capítulo 1 se presentan algunas herramientas de análisis que permiten describir y clasificar CPSMs. A través del Modelo formal (ver sección 1.1) es posible representar los componentes constitutivos de un CPSM (i.e. plataformas y servicios). Adicionalmente, en este Capítulo se definen los Parámetros característicos de un CPSM (ver sección 1.2). También es posible expresar, en términos del Modelo formal, las limitaciones que dichos parámetros se imponen mutuamente.
- En el Capítulo 2 se describen las actividades involucradas en el desarrollo de un CPSM. En la sección 2.1 se tratan las actividades que definen características generales del CPSM (e.g. lenguaje de programación, paradigma, propósito). En las secciones 2.2 y 2.3 se presentan las principales actividades que caracterizan a un CPSM: la definición de servicios provistos y la definición de plataformas compatibles con el CPSM, respectivamente. En la sección 2.4 se analizan diversos mecanismos de abstracción aplicados al desarrollo de un CPSM y algunos problemas de implementación asociados a dichos mecanismos.
- En el Capítulo 3 se presenta un caso de estudio que incluye el desarrollo de un CPSM, diseñado para sustentar un tipo de programa en particular. El Caso de Estudio se completa presentando un ejemplo de un programa sustentado sobre dicho CPSM. El ejemplo se introduce en tres intentos, cada uno de los cuales incorpora un nuevo servicio del CPSM.
- En el Capítulo 4 se describen algunos CPSMs utilizados en otros capítulos para ejemplificar diversos aspectos de un CPSM (e.g. parámetros característicos, actividades involucradas en el desarrollo de un CPSM, etc).
- En el Apéndice A se definen algunos términos cuyo significado puede interpretarse de diversas maneras según el contexto en que se encuentren. El objetivo de este Anexo es restar ambigüedad a los términos clave utilizados.

⁴En rigor, existen varias soluciones alternativas al problema planteado, entre ellas se cuentan: los lenguajes de programación interpretados (e.g. Perl, Python) y la ejecución de programas utilizando un *software* de emulación de entorno de ejecución (e.g. Wine, Cygwin). Este trabajo se centra en las soluciones que utilizan una capa de abstracción y poseen un único árbol de código fuente, con porciones de código comunes a todas las plataformas compatibles y porciones dependientes de la plataforma.

⁵El término *Middleware* frecuentemente connota una capa de abstracción sobre un sistema operativo de red[Tan02]. En esta tesis, el término se utiliza en un sentido amplio, denotando una capa de abstracción sobre un conjunto de plataformas, sin restringir su alcance a los sistemas operativos de red, ni su aplicación a los sistemas distribuidos.

⁶La interfaz provista por un *Middleware* puede dividirse en “servicios” (i.e. conjuntos de primitivas y recursos que colaboran para proveer una funcionalidad coherente). La interfaz de cada servicio provisto por un *Middleware* será denominada “interfaz abstracta del servicio”. En contraposición, la interfaz de programación de una plataforma (API), para un servicio, será la “interfaz nativa del servicio”.

1. Herramientas de análisis de un CPSM

A continuación se introducen dos herramientas que se utilizan a lo largo de este trabajo para analizar las características de un CPSM. La primer herramienta es un modelo formal de la estructura de un CPSM donde se reconocen sus principales componentes constitutivos y algunas restricciones impuestas sobre dichos componentes.

La segunda herramienta es un conjunto de parámetros que caracterizan a un CPSM. Inicialmente se presenta una descripción de cada parámetro por separado y luego se consideran las interacciones entre los mismos y las limitaciones que se imponen mutuamente. Para algunos de ellos, se presentan criterios de comparación que permiten evaluar un conjunto de CPSMs en relación al parámetro considerado.

En secciones posteriores se utiliza el modelo formal como marco de referencia para analizar las actividades involucradas en el desarrollo de un CPSM y comparar diversas implementaciones existentes de CPSMs utilizando los criterios derivados de los parámetros característicos presentados en la sección 1.2.

1.1. Modelo formal

En esta sección se presenta un modelo formal que permite analizar los problemas asociados con el desarrollo de un CPSM. Este modelo permite analizar las decisiones que implica el desarrollo de una capa de software que aisle los detalles de implementación dependientes de la plataforma. Adicionalmente, los elementos del modelo formal permiten derivar criterios de clasificación y análisis de CPSMs (ver sección 1.2).

El modelo formal definido a continuación se expresa en términos de álgebra de conjuntos, conectivas lógicas y cuantificadores, existenciales y universales. Esta terminología permite escribir de manera simbólica las relaciones entre los elementos constitutivos de un CPSM y comparar características de varios CPSMs. El objetivo del modelo es ilustrar la estructura de un CPSM y por lo tanto debe interpretarse como una descripción general del sistema y no como una construcción rigurosa en términos matemáticos.

El modelo puede definirse en los siguientes términos:

- Sea \mathcal{P} un conjunto de plataformas p^n , con $n = 1..N \in \mathcal{N}$.⁷
- Sea \mathcal{S} un conjunto de servicios s_m , con $m = 0..M \in \mathcal{N}$.
- Una notación para definir un CPSM, A , es:

$$A = CPSM(\mathcal{P}, \mathcal{S})$$

Es decir, A está definido como un CPSM que provee los servicios en \mathcal{S} , y es compatible con las plataformas en \mathcal{P} .

A continuación se presentan algunas definiciones adicionales que permiten caracterizar con mayor detalle el conjunto de servicios, \mathcal{S} :

- Sea i_m^n la interfaz del servicio $s_m \in \mathcal{S}$ existente en la plataforma $p^n \in \mathcal{P}$.
- Sea $\mathcal{I}_d(\mathcal{S}, \mathcal{P})$ el conjunto de interfaces dependientes de la plataforma, i_m^n .

⁷Donde \mathcal{N} es el conjunto de los números naturales.

- Sea i_m la interfaz abstracta⁸ del servicio $s_m \in \mathcal{S}$, es decir, la interfaz que provee CPSM para el servicio s_m , independientemente de la plataforma subyacente.
- Sea $\mathcal{I}(\mathcal{S})$ el conjunto de interfaces abstractas, i_m , independientes de la plataforma subyacente.
- Sea Σ un subconjunto de \mathcal{S} , cuyos elementos (servicios) existen en todas las plataformas p^n y sus respectivas interfaces son equivalentes⁹ en todas ellas:

$$\Sigma = \left\{ s_l \in \mathcal{S} \mid \forall a, b \in [1, N], \exists i_l^a \wedge \exists i_l^b \wedge i_l^a \equiv i_l^b \right\}$$

Es decir, Σ es el conjunto los servicios que son trivialmente compatibles con todas las plataformas consideradas en \mathcal{P} ¹⁰.

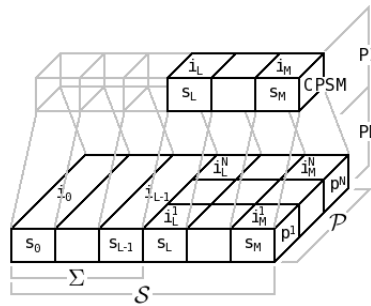


Figura 1.1: Diagrama genérico de CPSM

En la figura 1.1 se representa la estructura de un CPSM en términos de sus características formales. En el eje horizontal se grafican los servicios provistos (\mathcal{S}), en profundidad las plataformas soportadas (\mathcal{P}), y en el eje vertical la relación de abstracción entre capas, es decir, la relación de “uso” que una capa de software mantiene con sus capas adyacentes (i.e. superior e inferior). En esta dimensión se distinguen dos áreas, una región dependiente de la plataforma (indicada “PD”) y otra independiente de la plataforma (indicada “PI”). El objetivo de un CPSM no trivial¹¹ se ve representado en la transición entre el área dependiente de la plataforma (PD) y el área independiente de la plataforma (PI). Es decir, cualquier programa construido sobre (sustentado en) el CPSM, resulta contenido en la región PI y por lo tanto es independiente de las plataformas incluidas en \mathcal{P} . El trabajo necesario para aislar a dicho

⁸En lo sucesivo se utilizará de forma indistinta los términos “interfaz abstracta” e “interfaz independiente de la plataforma” para hacer referencia a la interfaz de un servicio que expone el CPSM a sus capas superiores.

⁹Un posible criterio de equivalencia de interfaces puede formularse de esta manera: dado un programa que utilice un servicio determinado, s_m , se dirá que las interfaces del servicio son equivalentes (respecto de dicho programa) en todas las plataformas de \mathcal{P} , si el programa compila en todas las plataformas y en todas ellas exhibe el mismo comportamiento en tiempo de ejecución (i.e., para cada conjunto de entradas, el programa devuelve el mismo conjunto de salidas en todas las plataformas). En estos términos, el criterio de equivalencia se define en relación a un programa. La rigurosidad de la equivalencia dependerá del grado de exhaustividad con que dicho programa utilice la interfaz del servicio.

¹⁰Entiéndase por “servicios trivialmente compatibles con un conjunto de plataformas”, a los servicios que se encuentran disponibles en dichas plataformas y que, en todas ellas, exponen interfaces nativas equivalentes (ver nota 9).

¹¹Mas adelante, en esta sección, se explicitan las características de un CPSM trivial.

programa de los detalles de implementación dependientes de la plataforma de los servicios que requiere, constituye la esencia del CPSM y su principal función.

Nótese el trazo gris que indica los servicios que inherentemente son independientes de la plataforma (aquellos incluidos en Σ). Resulta innecesario y redundante que el CPSM provea una interfaz para estos servicios, dada su naturaleza independiente de la plataforma¹².

El desarrollo de un CPSM requiere la existencia de, por lo menos, un servicio común a todas las plataformas de \mathcal{P} , con interfaces compatibles, es decir:

$$\exists s_0 \in \Sigma \text{ o } \#\Sigma \geq 1$$

Dicho de otro modo, si no existiera ningún punto de articulación entre dos plataformas, ni siquiera un lenguaje de programación en común¹³, no sería posible construir un CPSM compatible con ambas plataformas, ya que la selección de un lenguaje de programación constituiría una característica de dependencia de la plataforma. En lo sucesivo, este requerimiento será denominado *condición de existencia del punto de articulación*.

En caso de no cumplirse con la *condición de existencia del punto de articulación*, el tipo de desarrollo necesario para satisfacer las necesidades de independencia de la plataforma no es un CPSM sino un lenguaje de programación en común, y los artefactos a través de los cuales se llega a construir un software independiente de la plataforma no son otros que los compiladores de dicho lenguaje en cada plataforma soportada.

Un ejemplo de este escenario puede ser: dadas dos arquitecturas (p^A y p^B) con lenguajes de máquina distintos e incompatibles (i_0^A e i_0^B), donde no existe ningún lenguaje de programación compilable (y ejecutable) en ambas arquitecturas. En este caso, no es posible desarrollar un CPSM directamente. Sin embargo, para lograr abstracción de código fuente es posible desarrollar un lenguaje en común, s_0 (cuya definición estándar es i_0), que pueda compilarse al lenguaje de máquina particular de cada arquitectura, e implementar compiladores de dicho lenguaje en ambas arquitecturas (que en este caso deberían traducir programas escritos utilizando la definición estándar, i_0 , del lenguaje común, s_0 , al lenguaje de máquina propio de cada arquitectura, i_0^a , que luego de ser ensamblado con el ensamblador específico de la misma, daría lugar al ejecutable). Si bien el modelo formal es suficientemente amplio como para representar este caso (las arquitecturas conforman el conjunto \mathcal{P} , el lenguaje en común es el único elemento del conjunto \mathcal{S} , con su interfaz abstracta i_0 y los lenguajes de máquina de cada arquitectura constituyen las interfaces i_0^A e i_0^B), no se trata de un CPSM ya que los compiladores considerados son dos programas separados, distintos y escritos en diferentes lenguajes. En este caso, el punto de articulación es artificial ya que no está presente en el conjunto de programas realizados sino que es el compromiso existente en la estandarización del lenguaje en común. En contraposición, en un CPSM, el punto de articulación se manifiesta explícitamente a través de construcciones propias del lenguaje en el que esté escrito (e.g. clases abstractas, declaración de funciones, etc). Adicionalmente, este trabajo se restringirá a considerar las plataformas, \mathcal{P} ,

¹²Existen buenos argumentos a favor de generar interfaces para los servicios que son inherentemente independientes de la plataforma, aunque estos argumentos no se enfocan en proveer independencia de la plataforma, sino otro tipo de características deseables en un *Framework*, y particularmente en un CPSM, como ser la coherencia en las interfaces o la *Seguridad* (entendida como *safety*, ver sección 1.2)[Sch92].

¹³i.e. donde todo programa escrito en dicho lenguaje sea compilable en ambas plataformas sin necesidad de ser modificado y su ejecución exhiba un comportamiento similar. Nótese que la ausencia de un lenguaje en común implica también, que no existe un subconjunto útil (a los fines prácticos) de un lenguaje que cumpla con la condición antes mencionada.

en el sentido indicado en el Apéndice A, es decir, como Sistemas Operativos¹⁴.

En el caso más general, el punto de articulación, s_0 , es un lenguaje de programación estandarizado, con compiladores compatibles con el estándar del lenguaje, disponibles en todas las plataformas consideradas.

Un CPSM que es compatible con una única plataforma constituye un caso trivial, donde no se aporta independencia de la plataforma ya que sólo se considera una única plataforma compatible. En rigor, este tipo de pieza de software no debería denominarse CPSM ya que no cumple con el requisito principal de proveer independencia de la plataforma. Este caso se presenta para dar al lector una visión completa de la generalidad de un CPSM y cómo una pieza de software dependiente de la plataforma puede generalizarse a un CPSM compatible con una única plataforma. Esta generalización es el punto de partida para aumentar la *Compatibilidad* (ver sección 1.2) de la pieza de software dependiente de la plataforma, transformándola en un CPSM no trivial.

Un CPSM que sólo provee el servicio s_0 , es decir un CPSM para el cual $\mathcal{S} = \{s_0\}$ o, de forma más general, $\mathcal{S} = \Sigma$, es trivialmente compatible con todas las plataformas de \mathcal{P} y no aporta mayor independencia de la plataforma de la que originalmente existía.

Un caso más interesante son los programas para los cuales $\Sigma \subset \mathcal{S}$, es decir, cuando existe al menos un servicio, s_x perteneciente a \mathcal{S} y que no se encuentra en Σ y, por lo tanto, para el cual existen al menos dos plataformas donde las interfaces de dicho servicio son incompatibles:

$$s_x \in \mathcal{S} / \exists a, b \in [1, N], \exists i_x^a \wedge \exists i_x^b \wedge i_x^a \neq i_x^b$$

En vista de los elementos del modelo formal puede refinarse la definición de CPSM para eliminar los casos triviales y centrar la atención en el problema mencionado anteriormente:

Se desea un CPSM que garantice, a todo programa sustentado sobre el mismo, compatibilidad con las N plataformas de \mathcal{P} (con $N > 1$). Se requiere también que dicho CPSM provea al menos un servicio s_x disponible en las plataformas de \mathcal{P} pero no perteneciente a Σ , es decir, provea un servicio cuyas interfaces no son equivalentes en todas las plataformas de \mathcal{P} .

El objetivo ulterior es desarrollar un programa que encapsule las incompatibilidades de interfaz de todos los servicios $s_x \notin \Sigma$.

De la formalización precedente puede extraerse la siguiente conclusión: un programa que cumpla con las características enunciadas podrá clasificarse según las plataformas consideradas en \mathcal{P} , los servicios que provea en \mathcal{S} y las interfaces abstractas, i_m , de cada servicio s_m que exponga.

Como consecuencia, cabe destacarse que si el programa A es, en efecto, un CPSM, luego un programa B podría sustentarse directamente sobre las interfaces abstractas de A (es decir $\Sigma_B = \mathcal{S}_B = \mathcal{S}_A$), y adquirir compatibilidad automática con todas las plataformas con las que A es compatible ($\mathcal{P}_B = \mathcal{P}_A$).

De este análisis se desprenden tres desafíos al definir un CPSM: seleccionar las plataformas con las que será compatible, definir los servicios que serán provistos y especificar las interfaces correspondientes a cada servicio.

¹⁴Ulrich Drepper, en su artículo *Dictatorship of the Minorities*[Dre05] expone una postura radical respecto del soporte que, en su opinión, debería darse a diferentes arquitecturas en la biblioteca GNU del lenguaje C, *glibc*. El debate que propone puede analizarse en términos del modelo formal presentado en este trabajo, aunque ese análisis queda a cargo del lector.

Nótese que estos desafíos distan de ser triviales. Una elección demasiado estrecha en las plataformas compatibles reduce, en efecto, el alcance del programa¹⁵. Por otro lado, una elección demasiado ambiciosa podría reducir los servicios candidatos a ser incluidos en la capa de abstracción, debido a la existencia de, al menos, una plataforma en particular que no provea (y donde no sea posible construir) una implementación de alguna de las interfaces requeridas:

$$\exists p^n \in \mathcal{P}, s_m \in \mathcal{S} / \#i_m^n$$

Algo similar sucede con la selección de servicios y la especificación de sus interfaces. Si se provee gran cantidad de servicios, se reducirá el número de plataformas candidatas a ser incluidas en \mathcal{P} , debido a que disminuirá la cantidad de plataformas que proveen todos y cada uno de los servicios de \mathcal{S} . Si, por el contrario, se opta por reducir el número de servicios provistos, se compromete la funcionalidad del CPSM. Por ejemplo, un CPSM que provee un servicio de comunicaciones basado en *sockets* y, para aumentar el número de plataformas candidatas, expone únicamente primitivas bloqueantes, pero no provee un servicio de concurrencia. Este CPSM es de una utilidad muy reducida en casos donde se desea atender a más de un cliente por vez.

Respecto de la especificación de las interfaces de los servicios, si se opta por una interfaz minimalista, se cuenta con una mayor cantidad de plataformas candidatas a ser incluidas en el conjunto de plataformas compatibles, pero también se corre el riesgo de que se pierda la esencia del servicio debido a la ausencia de primitivas fundamentales del mismo. Por otro lado, si se busca una máxima expresividad¹⁶ de las interfaces, al igual que con el caso de selección de plataformas, se corre el riesgo de reducir el número de plataformas candidatas.

En este sentido, el desarrollo de un CPSM requiere de un delicado equilibrio entre distintos aspectos, como ser *Compatibilidad*, *Flexibilidad* y *Performance* (ver sección 1.2).

Por último, es posible realizar una simplificación que, sin pérdida de generalidad, permitirá escribir de manera más compacta (i.e. con menor cantidad de elementos del modelo) la relación entre los servicios trivialmente compatibles (en Σ) y aquellos que requieren un mecanismo de abstracción debido a que sus interfaces dependen de la plataforma. Puntualmente, dado que todos los servicios en Σ son trivialmente compatibles con todas y cada una de las plataformas de \mathcal{P} , se puede definir un único servicio s_Σ como la unión de todos los servicios de Σ :

$$s_\Sigma = \{s_0 \cup s_1 \cup \dots \cup s_{L-1}\}$$

Luego se puede redefinir Σ conteniendo como único elemento s_Σ :

$$\Sigma = \{s_\Sigma\}$$

Para mantener una coherencia con los subíndices se utilizará s_0 en lugar de s_Σ :

$$\Sigma = \{s_0\}$$

¹⁵Más aún, la elección de una única plataforma compatible a la hora de desarrollar un software es la reducción del problema a un caso trivial, donde, en efecto, se carece por completo de independencia de la plataforma.

¹⁶Debe entenderse por “expresividad de una interfaz”, la diversidad de operaciones que es posible realizar a través de ella. Por ejemplo la expresividad de una interfaz cuya única primitiva sea *int sumar2mas3()* es menor que la de otra interfaz cuya única primitiva sea *int sumar(int a, int b)*, dado que la segunda permite sumar cualquier par de números a y b, mientras que la primera sólo es capaz de calcular la suma entre 2 y 3. En lo sucesivo se utilizará el término “expresividad de una interfaz” en este sentido.

En estos términos, la cardinalidad de Σ , L , tendrá un valor unitario y el conjunto de los servicios que no son trivialmente compatibles se define como todos los servicios de \mathcal{S} que no se encuentran en Σ :

$$\mathcal{S} \cap \Sigma^C = \{s_1, s_2, \dots, s_M\}$$

y los servicios provistos, \mathcal{S} , pueden redefinirse, en términos de su compatibilidad, como aquellos servicios que son trivialmente compatibles con las plataformas en \mathcal{P} (i.e. $s_0 \in \Sigma$) y los servicios que dependen de la plataforma (i.e. $s_m \notin \Sigma$):

$$\mathcal{S} = \{s_0\} \cup \{s_1, s_2, \dots, s_M\}$$

De la formalización precedente debe notarse que, en lo sucesivo, se utilizará s_0 para referirse a todo el conjunto de servicios trivialmente compatibles y no sólo al primero de ellos. Este será el criterio adoptado a partir de aquí. La introducción de dos notaciones diferentes para los servicios que son trivialmente compatibles con las plataformas en \mathcal{P} (i.e. s_0 y $s_l \in \Sigma$) permite utilizar la notación compacta (s_0) cuando se desee centrar la atención en los servicios dependientes de la plataforma ($s_m \in \mathcal{S}$ y $\notin \Sigma$), mientras que cuando se desee individualizar las características de cada uno de los servicios trivialmente compatibles se puede optar por la notación explícita ($s_l \in \Sigma$).

1.2. Parámetros característicos de un CPSM

En esta sección se presenta una selección de parámetros que caracterizan, y a los que está sujeto, un CPSM. Estos parámetros permitirán evaluar cualitativamente el impacto de cada una de las actividades involucradas en el desarrollo de un CPSM. En este sentido, del análisis de un CPSM en relación a cada parámetro, es posible derivar un criterio de evaluación del CPSM relativo al parámetro en cuestión. Los parámetros considerados son: *Flexibilidad*, *Compatibilidad*, *Mantenimiento*, *Seguridad* y *Performance*.

Algunos de ellos encuentran una analogía directa con elementos definidos en el modelo formal precedente, como *Flexibilidad* y *Compatibilidad*, otros se asocian a características generales del CPSM, como ser *Mantenimiento*, *Seguridad* y *Performance*.

A continuación se describen los parámetros de forma aislada, más adelante se realizará un análisis de las relaciones y dependencias que existen entre ellos.

1.2.1. Flexibilidad

La *Flexibilidad* refleja el límite que el CPSM impone a la capacidad de generar funcionalidad (rutinas, programas, etc) de las capas superiores. Es decir, caracteriza la diversidad de implementaciones sustentables sobre el CPSM. El conjunto de servicios provistos, así como la expresividad de sus interfaces, son los principales factores en la determinación de la *Flexibilidad*[Tan95].

Un ejemplo cualitativo de este parámetro podría ser: dados dos CPSMs, donde ambos proveen un servicio de comunicaciones basado en *sockets*, si uno de ellos expone una interfaz a dicho servicio con soporte para comunicaciones con conexión, de tipo *streaming* (TCP), y para comunicaciones sin conexión, de tipo *datagram* (UDP), mientras que el otro CPSM no provee soporte para comunicaciones sin conexión, podrá afirmarse que el primer CPSM es más flexible que el segundo, dado que sobre el primero es posible sustentar un mayor número de tipos de programas, léase tanto aquellos que requieran comunicaciones con conexión cuanto aquellos que requieran comunicaciones sin conexión.

A continuación se presentan dos casos concretos que ilustran el ejemplo anterior, el código pertenece a NSPR¹⁷ (primer listado) y a NMSTL¹⁸ (segundo listado):

```

1  /* (...) */
2
3  /*
4  *****
5  * FUNCTION: PR_NewUDPSocket
6  * DESCRIPTION:
7  *   Create a new UDP socket.
8  * INPUTS:
9  *   None
10 * OUTPUTS:
11 *   None
12 * RETURN: PRFileDesc*
13 *   Upon successful completion, PR_NewUDPSocket returns a pointer
14 *   to the PRFileDesc created for the newly opened UDP socket.
15 *   Returns a NULL pointer if the creation of a new UDP socket failed.
16 *
17 *****
18 */
19
20 NSPR_API(PRFileDesc*)   PR_NewUDPSocket(void);
21
22 /*
23 *****
24 * FUNCTION: PR_NewTCPSocket
25 * DESCRIPTION:
26 *   Create a new TCP socket.
27 * INPUTS:
28 *   None
29 * OUTPUTS:
30 *   None
31 * RETURN: PRFileDesc*
32 *   Upon successful completion, PR_NewTCPSocket returns a pointer
33 *   to the PRFileDesc created for the newly opened TCP socket.
34 *   Returns a NULL pointer if the creation of a new TCP socket failed.
35 *
36 *****
37 */
38
39 NSPR_API(PRFileDesc*)   PR_NewTCPSocket(void);
40
41 /* (...) */

```

Listado 1.1: NSPR: prio.h

```

1  /* (...) */
2
3  /// A network Socket descriptor. This class will be expanded
4  /// to include Socket-specific I/O methods.
5  class Socket : public IOHandle {
6  public:
7      typedef enum {
8          none = 0,
9          nonblocking = 1,
10         acceptor = 2
11     } flags;
12     /* (...) */
13

```

¹⁷El Netscape Portable Runtime es el CPSM incluido dentro del código de los productos de Mozilla (e.g. Mozilla Firefox, Mozilla Thunderbird, etc).[Moz]

¹⁸Networking, Messaging, Servers, and Threading Library, un proyecto del grupo de Redes y sistemas móviles del laboratorio de ciencias de la computación del MIT.[Sal]

```
14 public:
15     /// Null constructor.
16     Socket() {}
17
18     /// Constructs a Socket from a given IOHandle.
19     Socket(const IOHandle& ioh) : IOHandle(ioh) {}
20
21     /// Updates the Socket error state if there's no current error
22     /// (useful after a connect). Returns true if there is no error
23     /// on the Socket.
24     Status stat() { /* (...) */ }
25
26     /// Returns the peer name of the Socket, if any.
27     Address getpeername() { /* (...) */ }
28
29     /// Returns the local name of the Socket, if any.
30     Address getsockname() { /* (...) */ }
31
32     /// Attempts to bind the Socket.
33     Status bind(const Address& a) { /* (...) */ }
34
35     /// Attempts to connect the Socket.
36     Status connect(const Address& a) { /* (...) */ }
37
38     /// Listens on the Socket.
39     Status listen(int backlog = 5) { /* (...) */ }
40
41     /// Attempts to accept a connection on the Socket.
42     Socket accept(Address& a) { /* (...) */ }
43 };
44
45 /* (...) */
46
47 /// A TCP Socket descriptor. This class will be expanded to include
48 /// TCP-Socket-specific I/O methods, e.g., SO_LINGER support.
49 class TCPSocket : public Socket {
50 public:
51     /// Constructs a TCP Socket.
52     TCPSocket() : Socket() {}
53     TCPSocket(Address addr, flags f = none) : Socket(AF_INET, SOCK_STREAM, addr, f)
54         {}
55     TCPSocket(const IOHandle& ioh) : Socket(ioh) {}
56 };
57 /* (...) */
```

Listado 1.2: NMSTL: net

Una aclaración respecto de los listados precedentes es que el segundo pertenece a NMSTL que no es un CPSM por no ofrecer ningún tipo de soporte para la independencia de la plataforma (i.e. NSMTL puede considerarse un CPSM trivial, ver sección 1.1). NMSTL es un desarrollo orientado a facilitar el acceso a los servicios de red en un ambiente Linux/Unix y por lo tanto, es posible comparar la interfaz que expone su servicio de redes con la interfaz expuesta por NSPR. En rigor, NSPR puede compararse con NMSTL si se lo reduce al caso trivial, es decir, si se lo circunscribe a las plataformas Linux/Unix.

Nótese que en el listado de NSPR se omitieron las primitivas de manipulación de *sockets* para facilitar la lectura. Tomando en cuenta esta consideración y analizando los tipos de servicio de comunicación (i.e. *streaming* con conexión, TCP, y *datagram* sin conexión, UDP) expuestos en los listados, es posible apreciar que la *Flexibilidad* de NSPR en el servicio de comunicaciones es mayor que la de NMSTL en tanto y en cuanto en NSPR es posible utilizar el tipo de servicio *datagram* sin conexión y en NMSTL no.

La *Flexibilidad* relativa puede definirse formalmente como sigue:

- Sean A y B dos CPSMs con sus respectivos conjuntos de servicios, \mathcal{S} :

$$\begin{aligned} A &= \text{CPSM}(\mathcal{P}, \mathcal{S}_a) \\ B &= \text{CPSM}(\mathcal{P}, \mathcal{S}_b) \end{aligned}$$

- Sea $F(x)$ la Flexibilidad del CPSM x
- Se cumple que¹⁹:

$$\text{Si } \mathcal{S}_a \supset \mathcal{S}_b, \text{ entonces } F(A) > F(B)$$

Un ejemplo de *Flexibilidad* relativa aplicada en otro contexto puede visualizarse en las primitivas *write* y *send* definidas en el estándar POSIX.1[IEE04]. A continuación se presentan los prototipos de ambas funciones y un extracto de la página de manual correspondiente a *send*²⁰:

```
1 ssize_t write(int fd, const void *buf, size_t count);
2
3 ssize_t send(int s, const void *buf, size_t len, int flags);
4
5 /* man 2 send:
6 "(...) The only difference between send() and write() is the presence of flags. With
7 zero flags parameter, send() is equivalent to write() (...) */
```

Listado 1.3: Interfaces POSIX de *write* y *send*

Como puede observarse, el manual indica que la única diferencia entre ambas funciones es la presencia del parámetro *flags*. Puesto que $\text{write}(fd, buf, count)$ se reduce a $\text{send}(fd, buf, count, 0)$, es decir, cualquier llamada a *write* se reduce a un caso especial de *send*, se sigue que la interfaz que expone *send* es más expresiva que *write*. Más aún, un CPSM que provea una abstracción de *send* pero no de *write* tendrá mayor expresividad, y por lo tanto mayor *Flexibilidad*, que el caso opuesto²¹. El ejemplo precedente ilustra de qué manera la expresividad de las interfaces influye en la *Flexibilidad* del CPSM.

1.2.2. Compatibilidad

La *Compatibilidad* es la característica que representa la capacidad de aislar efectivamente a los programas sustentados sobre el CPSM de la plataforma subyacente. Una posible medida de *Compatibilidad* está dada por la cardinalidad del conjunto de plataformas soportadas, \mathcal{P} . Esta medida podría refinarse ponderando las plataformas dentro del conjunto, según algún criterio²².

¹⁹Suponiendo que los servicios en común (i.e. en ambos conjuntos), poseen la misma especificación y por lo tanto la misma expresividad.

²⁰La página de manual corresponde a la sección 2 del “Linux Programmer’s Manual” en GNU/Linux 2.6.7.

²¹Esta afirmación es válida siempre y cuando para todas las plataformas consideradas en \mathcal{P} se cumpla la condición de reducción enunciada, donde *write* constituye un caso especial de *send*. Una excepción es Microsoft Windows donde la representación de un *socket* (SOCKET) difiere de un *file descriptor* (int). En esta plataforma, si se obtiene un *file descriptor* mediante *_open* y luego se intenta invocar *send* utilizándolo en lugar del argumento SOCKET, se obtiene un error 10038 cuya descripción es: “Socket operation on nonsocket. An operation was attempted on something that is not a socket. Either the socket handle parameter did not reference a valid socket, or for select, a member of an fd_set was not valid.”[Mic08e]

²²Por ejemplo: cantidad de usuarios estimados de esa plataforma, perfil de los potenciales usuarios del CPSM, modo de licenciamiento de la plataforma[Dre05], etc.

La *Compatibilidad* así definida hace referencia a las plataformas soportadas por un CPSM, es decir, aquellas respecto de las cuales se obtiene independencia al delegar en el CPSM el acceso a servicios que no son trivialmente independientes de la plataforma. A lo largo de este trabajo se utilizará el término en este sentido. Sin embargo, existen otros factores que pueden analizarse dentro de este parámetro como ser la compatibilidad de las interfaces binarias y la compatibilidad con compiladores específicos. Con el objeto de ilustrar estos dos casos de compatibilidad (que no serán tratados en este trabajo), se presenta una breve descripción de cada uno:

- El análisis de la compatibilidad de las interfaces binarias (denominadas *Application Binary Interfaces* o simplemente ABIs) consiste en evaluar la capacidad de invocar un código disponible en una biblioteca externa a un programa y cómo dicha biblioteca puede ser asociada a un proceso de forma implícita (en tiempo de *loading*) o explícita (en tiempo de ejecución). Sobre este tema existen muchas ramificaciones, entre ellas la forma en que distintos compiladores modelan los objetos, el esquema de *name decoration* (C) y *name mangling* (C++), etc. También es posible analizar hasta qué punto una biblioteca compilada con un compilador puede utilizarse en un programa compilado por otro compilador, las limitaciones de integración que impone el lenguaje de programación, etc.[Fog07]
- El análisis de la compatibilidad con compiladores específicos se enfoca en determinar qué construcciones de código son capaces de compilar determinados compiladores. Un CPSM que presenta construcciones que generan errores en algún compilador no será compatible con dicho compilador. En particular, compiladores con deficiencias en la implementación del estándar del lenguaje resultan problemáticos, ya que ciertas construcciones que se presuponían independientes de la plataforma, pueden generar errores de compilación (o lo que resulta más grave aún, comportamiento impredecible en tiempo de ejecución).

A continuación se presentan, como ejemplo de *Compatibilidad* las plataformas soportadas por el Netscape Portable Runtime (NSPR):

NSPR es compatible con la gran mayoría de plataformas basadas en Unix (incluyendo GNU/Linux y Mac OS X), Microsoft Windows, OS/2 y BeOS. Se compila y prueba en AIX, HP-UX, GNU/Linux, Mac OS X, Solaris, y Microsoft Windows regularmente[Moz06a].

Una posible definición de *Compatibilidad* relativa es:

- Sean A y B dos CPSMs con sus respectivos conjuntos de plataformas, \mathcal{P} :

$$\begin{aligned}A &= \text{CPSM}(\mathcal{P}_a, \mathcal{S}) \\ B &= \text{CPSM}(\mathcal{P}_b, \mathcal{S})\end{aligned}$$

- Sea $C(x)$ la Compatibilidad del CPSM x
- Se cumple que:

$$\text{Si } \mathcal{P}_a \supset \mathcal{P}_b, \text{ entonces } C(A) > C(B)$$

1.2.3. Mantenimiento

El *Mantenimiento* es la característica de poder evolucionar en los otros parámetros descritos en esta sección. En un CPSM de fácil *Mantenimiento* será más sencillo mejorar la *Compatibilidad*, es decir, agregar nuevas plataformas compatibles. Paralelamente, será más sencillo incrementar la *Flexibilidad* añadiendo nuevos servicios provistos.

En términos del principio de *Modular continuity*[Mey97], un CPSM de fácil *Mantenimiento* será aquel en el cual la propagación de las modificaciones resultantes del incremento de algún otro parámetro (e.g. *Flexibilidad* o *Compatibilidad*) se encuentre acotada²³.

El siguiente ejemplo permite visualizar cómo diferentes implementaciones de abstracción generan diferentes capacidades de *Mantenimiento*. El código que se presenta a continuación es una versión adaptada y simplificada de los ejemplos publicados en [Sch99b].

```
1 // Handle UNIX/Win32 portability differences.
2 #if defined (_WINSOCKAPI_)
3
4 #include <windows.h>
5 #pragma comment(lib, "ws2_32.lib")
6
7 #else
8
9 #include <sys/socket.h>
10 #include <netinet/in.h>
11
12 #endif /* _WINSOCKAPI_ */
13
14 #include <string.h>
15
16 // Main driver function. Some error handling has
17 // been omitted to save space in the example.
18 int main (int argc, char *argv[])
19 {
20     struct sockaddr_in sock_addr;
21
22     // Handle UNIX/Win32 portability differences.
23     #if defined (_WINSOCKAPI_)
24         SOCKET acceptor;
25     #else
26         int acceptor;
27     #endif /* _WINSOCKAPI_ */
28
29     // Create a local endpoint of communication.
30     acceptor = socket (PF_INET, SOCK_STREAM, 0);
31
32     // Set up the address to become a server.
33     memset (reinterpret_cast <void *> (&sock_addr),
34             0, sizeof sock_addr);
35     sock_addr.sin_family = AF_INET;
36     sock_addr.sin_port = htons (10000);
37     sock_addr.sin_addr.s_addr = htonl (INADDR_ANY);
38
39     // Associate address with endpoint.
40     bind (acceptor,
41          reinterpret_cast <struct sockaddr *>
42          (&sock_addr),
43          sizeof sock_addr);
44
45     // Make endpoint listen for connections.
```

²³En este sentido, una pequeña modificación en algún parámetro afecta a un pequeño conjunto de archivos. Se utiliza el término de “continuidad modular” como una analogía -poco rigurosa- con la continuidad de funciones matemáticas.

```
46 listen (acceptor, 5);
47
48 // Main server event loop.
49 for (;;) {
50     // Handle UNIX/Win32 portability differences.
51 #if defined (_WINSOCKAPI_)
52     SOCKET h;
53 #else
54     int h;
55 #endif /* _WINSOCKAPI_ */
56
57     // Block waiting for clients to connect.
58     h = accept (acceptor, 0, 0);
59
60     /* app logic */
61 }
62 /* NOTREACHED */
63 return 0;
64 }
```

Listado 1.4: Implementación de un servidor con compilación condicional

Nótese que si se desea mejorar la *Compatibilidad* del código (para que pueda ejecutarse en otras plataformas) es necesario agregar nuevas comparaciones en las dos líneas que contienen las instrucciones de preprocesamiento (*#if defined (_WINSOCKAPI_)*, etc). Una primera aproximación para mejorar la capacidad de *Mantenimiento* del código se presenta a continuación:

```
1 // Handle UNIX/Win32 portability differences.
2 #if defined (_WINSOCKAPI_)
3
4 #include <windows.h>
5 #pragma comment(lib, "ws2_32.lib")
6
7 #else
8
9 #include <sys/socket.h>
10 #include <netinet/in.h>
11 typedef int SOCKET;
12
13 #endif /* _WINSOCKAPI_ */
14
15 // Main driver function. Some error handling has
16 // been omitted to save space in the example.
17 int main (int argc, char *argv[])
18 {
19     struct sockaddr_in sock_addr;
20
21     SOCKET acceptor;
22
23     // Create a local endpoint of communication.
24     acceptor = socket (PF_INET, SOCK_STREAM, 0);
25
26     // Set up the address to become a server.
27     memset (reinterpret_cast <void *> (&sock_addr),
28             0, sizeof sock_addr);
29     sock_addr.sin_family = AF_INET;
30     sock_addr.sin_port = htons (10000);
31     sock_addr.sin_addr.s_addr = htonl (INADDR_ANY);
32
33     // Associate address with endpoint.
34     bind (acceptor,
35          reinterpret_cast <struct sockaddr *>
36          (&sock_addr),
37          sizeof sock_addr);
38 }
```

```
39 // Make endpoint listen for connections.
40 listen (acceptor, 5);
41
42 // Main server event loop.
43 for (;;) {
44     SOCKET h;
45
46     // Block waiting for clients to connect.
47     h = accept (acceptor, 0, 0);
48
49     /* app logic */
50 }
51 /* NOTREACHED */
52 return 0;
53 }
```

Listado 1.5: Reimplementación de un servidor con compilación condicional

En esta nueva versión sólo será necesario modificar un lugar que concentra toda la abstracción de la plataforma.

Una aproximación más elegante, resuelta con una versión adaptada [Sch99b] de los componentes del CPSM orientado a objetos ADAPTIVE Communication Environment (ACE) [Sch93] es:

```
1 /* ... */
2
3 // Main driver function. Some error handling has
4 // been omitted to save space in the example.
5 int main (int argc, char *argv[])
6 {
7     // Internet address of server.
8     INET_Addr addr (port);
9
10    // Passive-mode acceptor object.
11    SOCK_Acceptor server (addr);
12    SOCK_Stream new_stream;
13
14    // Wait for a connection from a client.
15    for (;;) {
16        // Accept a connection from a client.
17        server.accept (new_stream);
18
19        // Get the underlying handle.
20        SOCKET h = new_stream.get_handle ();
21
22        /* app logic */
23    }
24    /* NOTREACHED */
25    return 0;
26 }
```

Listado 1.6: Implementación adaptada de un servidor en ACE

En este último ejemplo, toda la abstracción se encuentra encapsulada dentro de ACE y el programa propiamente dicho deriva en este CPSM el acceso a los servicios de comunicaciones. Adicionalmente, el programa ha adquirido compatibilidad con un gran número de plataformas debido a que el CPSM subyacente se encarga de proveer esta compatibilidad.

Los ejemplos precedentes ilustran los problemas asociados al *Mantenimiento* de un programa con cierta independencia de la plataforma. Para analizar el *Mantenimiento* en el caso de un CPSM, se debe considerar el esfuerzo necesario para extender la *Compatibilidad* y/o

la *Flexibilidad* del mismo, agregando nuevas plataformas compatibles y/o nuevos servicios²⁴, respectivamente.

1.2.4. Seguridad

En esta sección es necesario hacer una aclaración respecto del término *Seguridad* y sus acepciones en inglés, para evitar posibles confusiones. En inglés, los términos *safety* y *security* suelen traducirse al castellano indistintamente como “seguridad”, a pesar de que su significado es claramente distinto aplicado al software. El término *safety* suele usarse en relación a la corrección de un programa, en el sentido de no presentar comportamientos inesperados debido a errores de programación, como por ejemplo *deadlocks*, acceso a posiciones de memoria inválidas, etc[BA90]. En cambio el término *security* se emplea en relación a las medidas necesarias para proteger un programa de posibles ataques (activos o pasivos) que puedan comprometer características del mismo (e.g. información, comportamiento, disponibilidad, etc.)[Sta06]. En este trabajo se utilizará el término *Seguridad* para hacer referencia a la primer acepción (i.e. la que se corresponde con *safety*).

La *Seguridad* puede entenderse en términos del criterio de *Modular protection*[Mey97]. Este criterio permite definir *Seguridad* como la capacidad de limitar la propagación de condiciones anormales en tiempo de ejecución.

En este sentido, la *Seguridad* es la capacidad de evitar errores comunes de programación debido a la utilización de un CPSM. Generalmente, la *Seguridad* se alcanza limitando las interfaces expuestas a los usuarios del CPSM, es decir, se especifican las interfaces abstractas de cada servicio (ver sección 2.2.1) intentando aislar al programador de construcciones peligrosas o potencialmente problemáticas. El impacto directo sobre las interfaces varía desde la aplicación de estrategias seguras de acceso a cierta funcionalidad hasta la supresión total de primitivas o parámetros en las mismas[Kem06].

Un ejemplo de aplicación de ambos criterios con el objeto de mejorar la *Seguridad* se ve en la especificación del servicio de concurrencia y sincronización provisto por las bibliotecas Boost. Este servicio en particular, denominado *Boost.Thread*, no presenta una abstracción de los *Event Objects* disponibles en Microsoft Windows²⁵. La justificación para la eliminar este tipo de primitiva de sincronización es su tendencia a producir errores de programación[Kem06, p. 33].

Adicionalmente, en *Boost.Thread*, no se proveen primitivas para obtener y liberar el control sobre un *Mutex* de forma explícita. En su lugar, se debe utilizar un patrón de *scoped-locking*[SSRB00] que asocia la sección crítica con el *scope* de un objeto. Si bien este patrón facilita el uso de variables *Mutex* para proteger una sección crítica, la inicialización de un objeto y su correspondiente destrucción en cada acceso a una sección crítica constituye un *overhead* que degrada la *Performance* general del programa. Por otro lado, la inexistencia de las primitivas *lock* y *unlock* de forma independiente impide del desarrollo de cierto tipo de programas, efectivamente disminuyendo la *Flexibilidad* del CPSM.

²⁴Dentro de esta categoría debe considerarse también, la extensión de la funcionalidad de un servicio mediante un incremento en la expresividad de su interfaz abstracta, ya sea a través de nuevas primitivas o de nuevos parámetros/valores en las mismas.

²⁵Los *Event Objects* se manipulan mediante funciones como *CreateEvent*, *OpenEvent*, *PulseEvent*, *ResetEvent* y *SetEvent*. Puntualmente, un *thread* puede bloquearse esperando un evento sobre un *Event Object* mediante *WaitForSingleObject*. Un *thread* bloqueado en un *Event Object* se desbloquea cuando otro *thread* invoca *SetEvent* o *PulseEvent* en dicho *Event Object*[Mic08d].

En el siguiente ejemplo (adaptado de [Sch92, p. 5]), se demuestran algunas fallas de *Seguridad* presentes en la interfaz de BSD Sockets en los sistemas POSIX. El código original presenta varios comentarios indicando los errores cometidos. En el ejemplo adaptado, se eliminaron dichos comentarios para dar la posibilidad al lector de intentar encontrar los errores por su cuenta. Luego del código se listan los errores para que pueda apreciarse cuán permeable a errores de programación resulta la interfaz utilizada.

```

1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4
5 const int PORT_NUM = 10000;
6 int main (void)
7 {
8     struct sockaddr_in s_addr;
9     int     length;
10    char     buf[1024];
11    int     s_fd, n_fd;
12
13    if (s_fd = socket (PF_UNIX, SOCK_DGRAM, 0) == -1)
14        return -1;
15
16    s_addr.sin_family = AF_INET;
17    s_addr.sin_port = PORT_NUM;
18    s_addr.sin_addr.s_addr = INADDR_ANY;
19
20    if (bind (s_fd, (struct sockaddr *) &s_addr,
21            sizeof s_addr) == -1)
22        perror ("bind"), exit (1);
23
24    if (n_fd = accept (s_fd, (struct sockaddr *) &s_addr, &length) == -1) {
25        int n;
26        while ((n = read (s_fd, buf, sizeof buf)) > 0)
27            write (n_fd, buf, n);
28        // Remainder omitted...
29    }
30    return 0;
31 }
```

Listado 1.7: Buggy Echo Server

Los errores cometidos en el código precedente son:

- Línea 16: no se inicializó en cero la estructura *s_addr* antes de utilizarla.
- Línea 16: la familia *AF_INET* es incompatible con el dominio del *socket PF_UNIX*.
- Línea 17: no se utilizó la función *htons()* para convertir el puerto al *byte-order* de la red.
- Línea 18: no se utilizó la función *htonl()* para convertir *INADDR_ANY* al *byte-order* de la red²⁶.
- Línea 20: la estructura *struct sockaddr_in* sólo debería utilizarse con *sockets* del dominio *PF_INET*.
- Línea 24: no se invocó la primitiva *listen()* para utilizar el *socket* en modo pasivo y por lo tanto no se verifican las precondiciones de la primitiva *accept()*.

²⁶Normalmente la constante *INADDR_ANY* se define como *0L*, por lo que resultaría innecesario convertir el *byte-order*, sin embargo, pueden existir plataformas donde esta suposición no sea cierta y por lo tanto es necesario asegurar la conversión mediante la función *htonl()*[The06].

- Línea 24: tanto *listen()*, cuanto *accept()* no deben utilizarse con *sockets* del tipo *SOCK_DGRAM*.
- Línea 24: no se inicializó el argumento *length* de la primitiva *accept()* para que contenga el tamaño de la estructura *s_addr*.
- Línea 24: la omisión de paréntesis en la expresión $n_fd = accept(...)$ resulta en que *n_fd* tendrá sólo dos valores posibles dependiendo de si el valor de retorno de *accept()* es igual a *-1* o no.
- Línea 26: se intenta leer de un *socket* en modo pasivo que sólo se debería utilizar para recibir conexiones.
- Línea 27: no se verifica que la cantidad escrita sea efectivamente *n* y no un valor menor debido al uso de *buffers (short-writes)*.

Si bien el programa precedente es un caso claramente patológico, salta a la vista que la interfaz utilizada no promueve la *Seguridad* de los programas sustentados sobre ella. Más aún, el programa precedente compilado con *gcc* versión 3.4.6 no genera ningún error ni emite ningún *warning* a no ser que se especifique la opción *-Wall*, en cuyo caso se detectan problemas sintácticos como la asignación sin paréntesis en la línea 24, pero no errores más sutiles como la incompatibilidad entre los argumentos de las primitivas, etc.

Mejorar la *Seguridad* de un CPSM tiende a comprometer otros parámetros como la *Flexibilidad* o la *Performance*. En el primer caso por ausencia de funcionalidad y en el segundo debido el *overhead* asociado a las estrategias adicionales necesarias para asegurar el correcto uso de funcionalidad que podría resultar peligroso en caso de no utilizarse correctamente.

1.2.5. Performance

La *Performance* de un CPSM es un aspecto que no debe dejarse de lado. Un programa sustentado sobre un CPSM sufrirá una degradación en la *Performance* en comparación con un programa similar, sustentado directamente sobre la API de una plataforma. Esta degradación se debe, principalmente, al nivel de indirección introducido al utilizar una capa de abstracción adicional[MDM⁺02]. Al desarrollar un CPSM se debe intentar minimizar el impacto que éste tiene sobre la *Performance*. Técnicas de metaprogramación[Bar05], así como el uso de *inlining*, pueden reducir ampliamente el impacto de un CPSM sobre la *Performance*. Por otro lado, la implementación en *userspace* de funcionalidad inexistente en alguna plataforma puede ser un importante factor de degradación de la *Performance*²⁷.

1.2.6. Relación entre los parámetros característicos de un CPSM

Los parámetros descritos en las secciones anteriores, y los criterios derivados a partir de estos, guardan una estrecha relación entre sí, y, en líneas generales, una modificación en uno

²⁷Nótese que existen casos donde una implementación en *userspace* puede ser más eficiente que la implementación provista por la plataforma. Algunos factores, como la reducción de *context-switching*, pueden mitigar el impacto sobre la *Performance* de implementaciones en *userspace*. Un ejemplo de este caso son los mecanismos de sincronización en Microsoft Windows, donde la sincronización a través de *CRITICAL_SECTIONs* (objetos en *userspace*) puede llegar a exhibir un aumento relativo de la *Performance* de hasta un orden de magnitud respecto a los *Windows Mutex Objects* (objetos en el *kernel*)[Bin08].

de ellos podría impactar sobre los otros. Por ejemplo, si se desea aumentar la *Compatibilidad* agregando una nueva plataforma, p^x , al conjunto \mathcal{P} , se debería verificar que estén disponibles en p^x todos los servicios soportados, \mathcal{S} y que sea posible implementar las interfaces definidas para cada servicio. Es decir:

$$\forall s_n \in \mathcal{S}, \exists i_n^x$$

Donde i_n^x es la interfaz (dependiente de la plataforma) del servicio s_n en la plataforma p^x .

En caso de no existir un servicio en p^x , se deberá modificar el conjunto de servicios soportados (degradando la *Flexibilidad*) o implementar toda la funcionalidad del mismo por encima de la plataforma. Esta segunda opción podría afectar la *Performance* del CPSM en dicha plataforma.

Simétricamente, aumentar la *Flexibilidad* incorporando un nuevo servicio s_x requiere verificar que, para todas las plataformas soportadas, \mathcal{P} , exista al menos una implementación (dependiente de la plataforma) de dicho servicio, es decir:

$$\forall p^m \in \mathcal{P}, \exists i_x^m$$

Donde i_x^m es la interfaz (dependiente de la plataforma) del servicio s_x en la plataforma p^m .

Un ejemplo de este caso se observa en la interfaz abstracta del servicio de comunicaciones de NSPR donde las primitivas bloqueantes reciben un parámetro de *timeout* para acotar la operación bloqueante. Sin embargo, la implementación de esta opción en Windows NT 3.5.1 tiene un comportamiento singular debido a la utilización de NT I/O Completion Ports[Mic06]. En la plataforma indicada, una llamada bloqueante se traduce en una operación de entrada/salida (I/O) asíncrona seguida de una suspensión del thread que realizó la invocación. Mientras tanto un thread de control consulta el Completion Port para obtener el resultado. En caso de un *timeout* se cancela la operación asíncrona mediante la primitiva *CancelIo()*. Sin embargo, en Windows NT 3.5.1 esta primitiva no está disponible y por lo tanto la única forma de cancelar la operación asíncrona pendiente es cerrando el *file descriptor* asociado a la operación. En este caso, el comportamiento singular en Windows NT 3.5.1 compromete la independencia de la plataforma del CPSM. Dada la madurez del CPSM, modificar la interfaz abstracta del servicio no es una opción y por lo tanto quedan dos alternativas para solucionar el problema: eliminar la *Compatibilidad* con Windows NT 3.5.1 o utilizar la versión de NSPR optimizada para Windows 32-bit (que no utiliza I/O Completion Ports) resignando la *Performance* alcanzable en la versión optimizada para Windows NT[His04].

En la sección 1.2.4 se mencionó el caso de *Boost.Thread* donde una decisión de diseño fue adoptar como característica predominante la *Seguridad* por sobre la *Flexibilidad*. Puntualmente, esto se refleja en la ausencia de primitivas de bloqueo explícito de *Mutex* en favor de una implementación basada en el patrón *scoped-locking*[SSRB00].

Del análisis precedente es posible concluir que los parámetros que caracterizan a un CPSM deben ajustarse de forma tal que permitan un equilibrio entre la funcionalidad provista (*Flexibilidad*) y las plataformas soportadas (*Compatibilidad*). Adicionalmente, es posible observar el impacto que una decisión incorrecta (o muy ambiciosa) en los demás parámetros, puede tener sobre la *Performance* del CPSM.

Respecto de la *Seguridad*, se debe hacer una última aclaración. El análisis presentado en este trabajo se enfoca en las características propias de un CPSM en lo que respecta a proveer independencia de la plataforma subyacente. Sin embargo, el desarrollo de un CPSM es un buen punto de partida para generar un *framework* que mejore la *Seguridad* de los programas sustentados sobre él. Esta característica no es exclusiva de CPSMs. El desarrollo NMSTL es un

claro ejemplo de una API que no provee independencia de la plataforma pero que está orientada a mejorar la *Seguridad* de (un subconjunto de) la interfaz de BSD Sockets[Sal]. Existen CPSMs donde se privilegia la *Flexibilidad* por sobre la *Seguridad* y por lo tanto se intenta exhibir una interfaz lo más parecida posible a la interfaz nativa²⁸ de cada plataforma. Este es el caso de NSPR[Moz]. En el otro extremo, es posible mencionar CPSMs como ACE que no sólo proveen independencia de la plataforma, sino también un importante juego de patrones y *wrappers* orientados a mejorar la *Seguridad* sin comprometer la *Performance*[Sch99b].

En la siguiente tabla se resume la comparación:

Nombre	Independencia de la Plataforma	Énfasis en la <i>Seguridad</i>
NMSTL	NO	SI
NSPR	SI	NO
ACE	SI	SI

1.3. Resumen

Modelo formal (sección 1.1):

- Un CPSM es compatible con un conjunto de plataformas que, en el modelo formal, se expresan mediante el conjunto \mathcal{P} , cuyos elementos se expresan como p^n con $n \in [1, N]$.
- Los servicios que provee el CPSM se expresan mediante el conjunto \mathcal{S} , cuyos elementos se expresan como s_m con $m \in [0, M]$.
- El servicio representado mediante el símbolo s_0 constituye el punto de articulación del CPSM, es decir, el servicio s_0 está compuesto por todas las interfaces que son inherentemente independientes de la plataforma para todas las plataformas incluidas en \mathcal{P} . (La cardinalidad del conjunto de servicios, \mathcal{S} se define como $M + 1$ para indicar la presencia del servicio s_0 y M servicios cuya implementación depende de la plataforma).
- El servicio s_0 es el único elemento del subconjunto Σ de \mathcal{S} , que representa a todos los servicios que son inherentemente independientes de la plataforma para el conjunto de plataformas \mathcal{P} . (El elemento Σ se introduce para poder generalizar el modelo formal al caso en el que existe más de un servicio inherentemente independiente de las plataformas en \mathcal{P} y por claridad resulte conveniente individualizar estos servicios en lugar de concentrarlos en el elemento s_0).
- La *condición de existencia del punto de articulación* afirma que en todo CPSM debe existir el servicio s_0 , o en términos de Σ , que el conjunto Σ debe contener al menos un elemento.
- Para cada servicio provisto por el CPSM, es decir, para cada elemento s_m perteneciente a \mathcal{S} , el CPSM expone una interfaz independiente de la plataforma simbolizada como i_m , siendo $\mathcal{I}(\mathcal{S})$ el conjunto de todas las interfaces independientes de la plataforma, correspondientes con cada servicio de \mathcal{S} .

²⁸i.e. interfaz dependiente de la plataforma.

- Para cada interfaz independiente de la plataforma, i_m debe existir (o debe ser factible realizar) una implementación dependiente de la plataforma, i_m^n para cada una de las N plataformas incluidas en \mathcal{P} .
- Se reconocen dos casos triviales para un CPSM:
 - Un CPSM que soporta una única plataforma (reduciendo el CPSM a una capa intermedia que NO provee independencia de la plataforma, como el caso de NMSTL), simbolizado en el modelo formal como: $\#\mathcal{P} = 1$.
 - Un CPSM que provee únicamente servicios que son inherentemente independientes de las plataformas en \mathcal{P} , simbolizado en el modelo formal como: $\mathcal{S} \equiv \Sigma$ o $\mathcal{S} = \{s_0\}$.²⁹
- Como consecuencia, el caso no trivial analizado en este trabajo es el de los CPSMs compatibles con más de una plataforma ($\#\mathcal{P} > 1$) y con al menos un servicio que no es inherentemente independiente de la plataforma, para las plataformas incluidas en \mathcal{P} ($\exists s_m \in \mathcal{S} / s_m \notin \Sigma$).

Parámetros característicos de un CPSM (sección 1.2):

- La *Flexibilidad* es la característica asociada a la disponibilidad de servicios (\mathcal{S}) y la expresividad de las interfaces abstractas de los mismos ($\mathcal{I}(\mathcal{S})$). A mayor *Flexibilidad* mayor número de servicios, y/o mayor número de primitivas en las interfaces abstractas, y/o mayor cantidad de opciones disponibles en dichas primitivas.
- La *Compatibilidad* se asocia a las plataformas compatibles con el CPSM (\mathcal{P}), y en las cuales se encapsula la implementación, dependiente de la plataforma, de cada uno de los servicios provistos (\mathcal{S}) con el objeto de que estos puedan ser utilizados independientemente de la plataforma subyacente.
- El *Mantenimiento* caracteriza la capacidad de evolucionar del CPSM incrementando alguno(s) de los otros parámetros (e.g. *Flexibilidad*, *Compatibilidad*) y la complejidad asociada a esta evolución.
- La *Seguridad* es la capacidad de evitar errores comunes de programación debido a ambigüedades existentes en las interfaces, secuencias de invocación implícitas o no muy claras, etc. (El término inglés correspondiente a *Seguridad* es *safety* y no debe confundirse con la seguridad en sentido de *security*, en la sección 1.2.4 se explica la diferencia entre estos términos).
- La *Performance* de un programa sustentado sobre un CPSM puede degradarse debido al nivel de indirección introducido al utilizar una capa de abstracción adicional. Al desarrollar un CPSM se debe intentar minimizar el impacto que éste tiene sobre la *Performance* a través de técnicas de metaprogramación y uso de *inlining*.

²⁹Entiéndase por equivalencia, que ambos conjuntos contienen los mismos elementos

2. Desarrollo de un CPSM

En este capítulo se describen las principales actividades involucradas en el desarrollo de un CPSM. Las actividades se presentan organizadas de la siguiente manera:

Inicialmente se presentan actividades de carácter general que contemplan decisiones de fondo en el desarrollo, como ser el paradigma, el lenguaje de programación y el propósito del CPSM.

Luego se presentan las actividades relacionadas con la definición de las características esenciales de un CPSM, a saber: las plataformas compatibles y los servicios que provee.

Por último se presentan las actividades relacionadas con la implementación de los mecanismos que posibilitan realizar la abstracción efectiva de la plataforma.

2.1. Características generales de un CPSM

Las actividades asociadas a las características generales de un CPSM son aquellas que determinan el marco tecnológico en el cual se construye el CPSM.

En las secciones subsiguientes se presentan las actividades de *Selección de paradigma*, *Selección del lenguaje de programación* y *Propósito del CPSM*, en relación al desarrollo de un CPSM.

2.1.1. Selección de paradigma

El paradigma bajo el cual se construye un CPSM es una característica muy importante del mismo. Existen diversos paradigmas que pueden considerarse para desarrollar un CPSM, y la elección de uno en particular tendrá una serie de ventajas y desventajas comparativas respecto de los otros.

La *Selección del lenguaje de programación* (ver sección 2.1.2) está estrechamente asociada a la *Selección de paradigma*. Dependiendo del orden en el que se tomen las decisiones o la jerarquía de las mismas, se deberá hablar de una dependencia entre el paradigma y el lenguaje o viceversa. Es decir, si el lenguaje de programación es impuesto, la selección de paradigma se verá reducida a los paradigmas soportados por dicho lenguaje. Por ejemplo: si el lenguaje es C++ u Object Pascal, se podrá optar por un paradigma estructurado o uno orientado a objetos, si el lenguaje es Java, la única opción posible será un paradigma orientado a objetos, etc. Análogamente, si se impone un paradigma, el conjunto de lenguajes candidatos se verá limitado a aquellos compatibles con dicho paradigma. Por ejemplo, si el paradigma es orientado a objetos, se deberían descartar los lenguajes estructurados tradicionales.

Nótese que la selección de paradigma impacta directamente en la *Definición de servicios provistos por el CPSM* (ver sección 2.2), ya que las operaciones definidas en las interfaces y su interacción, así como el posible estado persistente entre invocaciones, dependerá del paradigma seleccionado. Por ejemplo, un paradigma orientado a objetos fomenta, actualmente, el uso de diagramas UML para especificar las interfaces abstractas y su comportamiento (diagramas de interacción). Por otro lado, en un paradigma estructurado podría ser suficiente utilizar pre y post condiciones en cada función y/o procedimiento, descripción de parámetros, valor de retorno, etc.

Algunas ventajas y desventajas comparativas del paradigma estructurado vs. el paradigma orientado a objetos en lo que respecta al desarrollo de un CPSM se mencionan a continuación:

- Compatibilidad binaria: para lenguajes de programación como C/C++, la selección de paradigma trae aparejada una importante limitación en la compatibilidad binaria que podrá ofrecer el CPSM. En la sección 1.2.2 se comentaron algunos de los problemas asociados a la compatibilidad entre ABIs. Puntualmente, en la actualidad las implementaciones de compiladores de C/C++ manifiestan grandes discrepancias en lo que respecta a la representación binaria de las estructuras asociadas a la programación orientada a objetos. Estas discrepancias se ponen de manifiesto en la diversidad de esquemas de *name mangling* que se utilizan para generar los nombres de los símbolos exportados en las bibliotecas dinámicas compiladas por diferentes compiladores[Fog07].

En el Listado 2.1 se presenta un ejemplo de diferencias en los esquemas de *name mangling* para dos compiladores. El código fue compilado en Windows 2003 Server utilizando dos compiladores de C/C++ diferentes, ambos disponibles en dicha plataforma. Luego de compilar el código, se analizaron los símbolos exportados por cada biblioteca resultante mediante el programa *Dependency Walker*³⁰.

En la Figura 2.1 se presenta el resultado de compilar el código con g++.exe (GCC) 3.4.2 (mingw-special) dentro del entorno MSYS.

En la Figura 2.2 se presenta el resultado de compilar el código con el Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8804 for 80x86 (también conocido como cl.exe).

```
1 #ifdef WIN32
2 #include <windows.h>
3 BOOL APIENTRY DllMain( HANDLE hModule,
4                       DWORD ul_reason_for_call,
5                       LPVOID lpReserved)
6 {
7     return TRUE;
8 }
9 #define EXPORT_TEST __declspec(dllexport)
10 #else
11 #define EXPORT_TEST
12 #endif // WIN32
13
14 #include <stdio.h>
15 class EXPORT_TEST Test
16 {
17     const char* hello();
18 };
19
20 EXPORT_TEST const char* Test::hello()
21 {
22     return "Hello World";
23 }
```

Listado 2.1: Ejemplo de biblioteca que exporta una clase

Las Figuras 2.1 y 2.2 ponen de manifiesto las diferencias en los esquemas de *name mangling* de los dos compiladores utilizados. Como conclusión del ejemplo precedente, es posible apreciar que una biblioteca en C++ compilada con uno de los compiladores mencionados no será compatible con un programa (u otra biblioteca) compilado con el otro compilador, dado que no se podrán resolver los símbolos exportados, puesto que los esquemas de *name mangling* difieren. En el caso del lenguaje de programación C tradicional,

³⁰Este programa puede descargarse gratuitamente de <http://www.dependencywalker.com/>

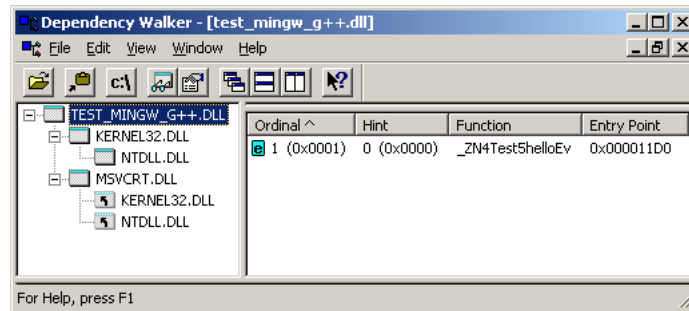


Figura 2.1: Símbolos exportados luego de compilar el código del Listado 2.1 con g++

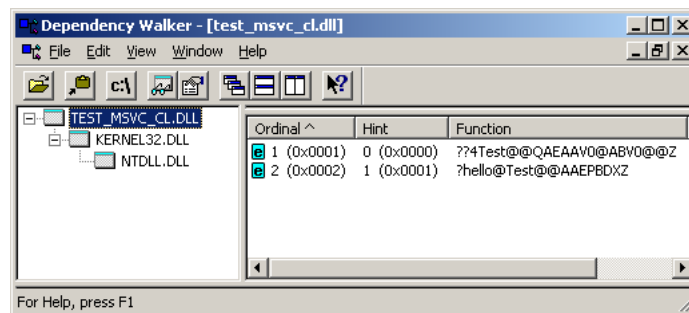


Figura 2.2: Símbolos exportados luego de compilar el código del Listado 2.1 con cl.exe

los símbolos se exportan a través de un esquema de *name decoration* que puede deshabilitarse³¹ permitiendo combinar bibliotecas compiladas por diferentes compiladores.

- *Seguridad*: el paradigma orientado a objetos resulta más adecuado para garantizar la *Seguridad* de un CPSM, debido a características como la posibilidad de ocultar el estado de los objetos y la existencia de constructores y destructores que permiten inicializar y liberar correctamente los recursos asociados a los objetos. El patrón de *scoped locking*[SSRB00] es un claro ejemplo de aplicación de las características básicas de la programación orientada a objetos para mejorar la *Seguridad* de la interfaz de un servicio de sincronización.
- *Abstracción y Mantenimiento*: el paradigma orientado a objetos permite explotar algunas características como la herencia y el polimorfismo para facilitar la abstracción de la plataforma subyacente. Por ejemplo, es posible definir las interfaces abstractas de un servicio como una clase abstracta y luego generar una implementación concreta de dicha clase abstracta para cada plataforma.

En la Figura 2.3 se muestra un ejemplo de una interfaz abstracta para un servicio de comunicaciones, representado a través de una clase abstracta, y sus dos implementaciones (*BSDSock* y *WINSock*) representadas por subclases concretas.

La especificación de direcciones de red en *BSD Sockets* es un ejemplo de una abstracción

³¹Aún en C++ es posible deshabilitar el esquema de *name decoration* de algunas funciones estáticas (mediante la construcción *extern "C" ...*[Mar06, How to mix C and C++]), pero debido a la complejidad asociada a la representación del paradigma orientado a objetos, no es posible deshabilitar el esquema de *name mangling*[Fog07].

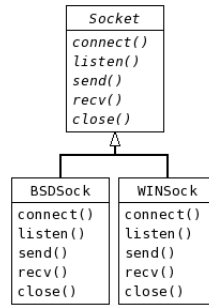


Figura 2.3: Abstracción en un servicio mediante herencia

primitiva de implementaciones alternativas (ver sección 2.4.1) en el lenguaje de programación C. Puntualmente, las direcciones de red se representan, de manera abstracta, mediante una variable del tipo *struct sockaddr*. Las implementaciones (alternativas) concretas de direcciones de red para diferentes familias de protocolos, se representan mediante variables que “extienden” la definición de campos del tipo *struct sockaddr*. Por ejemplo, las direcciones para la familia *AF_INET* se representan mediante las variables del tipo *struct sockaddr_in*[IEE04]. El uso de este tipo de abstracción en C, trae aparejada la necesidad de utilizar *typecasts* para transformar las variables y sumado a eso, las funciones que reciben estructuras de este tipo requieren que se indique explícitamente el tamaño de la variable. Esto compromete la *Seguridad* de la interfaz. En la sección 1.2.4, se presenta un ejemplo donde estos problemas se ponen de manifiesto (ver Listado 1.7).

2.1.2. Selección del lenguaje de programación

La selección del lenguaje de programación en el cual se desarrollará un CPSM suele no ser una decisión libre, ya que generalmente la construcción de un CPSM se ve subordinada a soportar el desarrollo de otros programas. Más aún, en algunos casos, el código de abstracción de la plataforma no tiene identidad individual y se encuentra fusionado con el programa al cual debe aislar de las plataformas subyacentes³².

Al margen de las restricciones que son impuestas sobre los lenguajes de programación candidatos³³ debido al contexto en el que se desarrolla el CPSM (programas existentes en un lenguaje dado, desarrollo a futuro de programas en un lenguaje seleccionado arbitrariamente, etc), deben considerarse las limitaciones derivadas de otras actividades involucradas en el desarrollo de un CPSM.

En primer lugar, la *Definición de plataformas compatibles con el CPSM* (ver sección 2.3) limita los lenguajes candidatos a aquellos que cumplan con la restricción de existencia de un punto de articulación (ver sección 1.1), i.e. que estén disponibles en todas las plataformas compatibles con el CPSM.

A la vez, la *Definición de servicios provistos por el CPSM* (ver sección 2.2) impone la

³²Este es el caso del soporte que se provee en la Java HotSpot Performance Engine[Sun08]

³³Los lenguajes de programación candidatos son aquellos que, dadas sus características y su disponibilidad en las distintas plataformas compatibles con el CPSM, pueden ser seleccionados para desarrollar el CPSM. Del conjunto de lenguajes de programación candidatos se deberá seleccionar un lenguaje de programación que formará parte del conjunto de servicios trivialmente independientes de la plataforma, Σ , puesto que será el lenguaje en el cual se programe el CPSM.

limitación adicional de que se cuente con implementaciones (potencialmente incompatibles) de todos los servicios incluidos en \mathcal{S} . Esta restricción debe entenderse en sentido amplio, es decir, en caso de no existir una implementación, o de existir una implementación que no sea suficientemente expresiva como para ser adaptada a la interfaz abstracta (ver sección 2.2.1) de un servicio, debe ser posible implementar la funcionalidad faltante en el lenguaje seleccionado. En algunos casos, realizar una implementación en *userspace* puede ser la única solución que permita cubrir la ausencia de implementación de un servicio en una plataforma en particular, pero debe tenerse especial cuidado con el impacto que este tipo de implementación tendrá sobre la *Performance* del servicio en dicha plataforma. En el Caso de Estudio (ver Capítulo 3), se presenta un ejemplo de este escenario al implementar *Condition Variables* en Microsoft Windows (ver Listado 3.28).

Teniendo en cuenta las limitaciones presentadas, el conjunto de lenguajes de programación candidatos se reduce a aquellos que estén disponibles en todas las plataformas³⁴ con las que el CPSM debe ser compatible (\mathcal{P}), para los cuales exista, o sea factible desarrollar, una implementación de cada servicio provisto por el CPSM (\mathcal{S}).

Formalmente:

- Sea $\mathcal{L}^n(\mathcal{S})$ el conjunto de lenguajes de programación disponibles en la plataforma p^n en los cuales existe, o es factible desarrollar, una implementación para cada servicio en \mathcal{S} .
- Sea $\mathcal{L}_c(\mathcal{P}, \mathcal{S})$ el conjunto de lenguajes de programación candidatos para desarrollar un CPSM que provea los servicios \mathcal{S} y sea compatible con las plataformas \mathcal{P} .

Entonces,

$$\mathcal{L}_c(\mathcal{P}, \mathcal{S}) = \{\mathcal{L}^1(\mathcal{S}) \cap \mathcal{L}^2(\mathcal{S}) \cap \dots \cap \mathcal{L}^N(\mathcal{S})\}$$

Al seleccionar un lenguaje del conjunto de lenguajes candidatos, éste, automáticamente forma parte del conjunto de servicios trivialmente compatibles con todas las plataformas.

Formalmente:

$$\text{Sea } l_c \in \mathcal{L}_c(\mathcal{P}, \mathcal{S}) \text{ el lenguaje seleccionado, entonces } l_c \in \Sigma$$

Por otra parte, si no existe un lenguaje candidato que cumpla con todas las restricciones enunciadas (i.e. no se cumple con la *condición de existencia del punto de articulación*), no será posible construir un CPSM y se deberá analizar cuál de las actividades involucradas en el desarrollo de un CPSM (e.g. la *Definición de plataformas compatibles con el CPSM* y/o la *Definición de servicios provistos por el CPSM*), se debería acotar para garantizar la existencia de, al menos, un lenguaje de programación candidato.

$$\text{Si dados } \mathcal{P} \text{ y } \mathcal{S}, \nexists l_c \in \mathcal{L}_c(\mathcal{P}, \mathcal{S}), \text{ entonces } \nexists \text{CPSM}(\mathcal{P}, \mathcal{S})$$

Por ejemplo, un CPSM que define un servicio de Administración de procesos que incluye una primitiva similar a *fork()*[IEEE04], y dicho CPSM debe ser compatible con Microsoft Windows. Dado que la API de Windows, en el lenguaje de programación C, carece de una primitiva similar a *fork()*, en principio, no será posible desarrollar el CPSM en dicho lenguaje. En este ejemplo, debería evaluarse si es factible implementar en *userspace* esta funcionalidad para algún lenguaje de programación y en caso de concluirse que no es factible, se deberán reconsiderar las características del CPSM.

³⁴Entiéndase por “lenguaje de programación disponible en una plataforma” a los lenguajes de programación para los cuales es posible desarrollar un programa en (y para) la plataforma en cuestión.

2.1.3. Propósito del CPSM

El propósito de un CPSM puede entenderse como la intención con la cual fue construido. Es decir, el propósito de un CPSM caracteriza y limita a los programas que podrían sustentarse sobre el mismo. En particular se pueden distinguir dos categorías dentro de las cuales ubicar a un CPSM en función de su propósito: CPSM de propósito general, y CPSM de propósito específico.

Dependiendo del propósito para el cual es desarrollado, un CPSM se enfoca en una característica predominante en perjuicio de otras características posibles. En términos de los parámetros introducidos en la sección 1.2, el propósito determina el tipo de interfaces abstractas que expondrá el CPSM.

Se pueden apreciar notables diferencias de construcción al comparar CPSMs construidos con propósitos diferentes. Entre estas diferencias, se pueden mencionar la disponibilidad de servicios, la expresividad de las interfaces de cada servicio, la compatibilidad con determinadas plataformas, etc.

Por ejemplo, al comparar dos CPSM de propósito específico, uno de ellos orientado a garantizar independencia de la plataforma en programas cliente-servidor, y el otro centrado en multimedia, se espera obtener grandes diferencias. El primer CPSM deberá proveer servicios de Comunicaciones y Concurrencia, mientras que el segundo puede prescindir de las Comunicaciones, pero deberá proveer soporte avanzado para entrada y salida (eventos, periféricos, etc), audio y video. Dos CPSMs existentes que ilustran el ejemplo precedente son NSPR y SDL³⁵.

Al comparar dos CPSMs de propósito diferente como NSPR y SDL, es esperable que el número de servicios en común resulte muy inferior al número de servicios que cada uno provee:

$$\# \{ \mathcal{S}_{NSPR} \cap \mathcal{S}_{SDL} \} \ll \min(\# \mathcal{S}_{NSPR}, \# \mathcal{S}_{SDL})$$

Es decir, existe un número muy pequeño de servicios en común entre NSPR y SDL, en relación con los servicios provistos tanto por NSPR cuanto por SDL. En el ejemplo³⁶:

$$\begin{aligned} \mathcal{S}_{SDL} &= \{ \text{Concurrencia, Multimedia (Video, Audio, Joystick, CD-ROM),} \\ &\quad \text{Timers, Endian independence, Window Management, Bibliotecas} \\ &\quad \text{dinámicas} \} \\ \mathcal{S}_{NSPR} &= \{ \text{Concurrencia, Comunicaciones, Filesystem, Bibliotecas dinámicas,} \\ &\quad \text{Comunicación entre procesos, Administración de procesos, Timers,} \\ &\quad \text{Fecha/Hora, Manejo de memoria, ...} \} \end{aligned}$$

$$\# \{ \mathcal{S}_{NSPR} \cap \mathcal{S}_{SDL} \} = 3 \text{ (Concurrencia, Timers, Bibliotecas dinámicas)}$$

$$\min(\# \mathcal{S}_{NSPR}, \# \mathcal{S}_{SDL}) = \# \mathcal{S}_{SDL} = 6$$

Una comparación de esta índole no resulta de gran interés debido a que dos CPSMs diseñados para cumplir con distintos propósitos tendrán por fuerza que proveer diferentes servicios y con un grado de expresividad acorde al propósito que deben satisfacer.

³⁵Simple DirectMedia Layer es un CPSM orientado al desarrollo de programas con alto contenido de multimedia, en especial videojuegos[Lan05].

³⁶En la sección 2.2 se describen, de manera más detallada, algunos de los servicios utilizados en la comparación.

Un caso más interesante son los CPSM de propósito general, donde el diseño del CPSM debe responder a un amplio espectro de necesidades (i.e. debe proveer un amplio espectro de servicios).

Al contrastar las características predominantes en un CPSM de propósito general con las de uno de propósito específico se pueden obtener algunas conclusiones interesantes relacionadas con impacto que derivará sobre la especificación de interfaces (ver sección 2.2.1), debido al propósito del CPSM.

Los CPSM de propósito específico suelen centrarse en proveer interfaces suficientemente expresivas para contemplar todos los casos de uso asociados a dicho propósito. Es decir, en proveer buena *Flexibilidad* y *Performance* en su campo específico de acción. Adicionalmente, pueden enfocarse en proveer *Compatibilidad* con un conjunto de plataformas determinado.

El caso de los CPSM de propósito general es diametralmente opuesto. Debido a la ausencia de los requerimientos de *Flexibilidad* y *Compatibilidad* que impone un propósito específico, este tipo de CPSM suele especificar interfaces cuya característica principal es la *Seguridad*[Kem06], sin vulnerar, cuando sea posible, la *Performance*.

Boost, un CPSM de propósito general, tiende a predominar la *Seguridad* frente a la *Flexibilidad* o la *Performance*. Motiva esta decisión, la premisa de disminuir al mínimo los posibles errores de programación debidos a falta de experiencia del programador o la complejidad del programa. Por otro lado, los CPSMs de propósito específico, como NSPR y SDL, tienden a predominar *Flexibilidad* por sobre otros parámetros.

A continuación se contraponen estas dos características predominantes con sus ventajas y sus desventajas, a saber: *Flexibilidad* y *Seguridad*. No se descarta la posibilidad de contraponer otros parámetros, sin embargo la selección de *Flexibilidad* vs *Seguridad* se basa en el hecho de que actualmente los CPSM tienden a orientarse hacia una de estas dos características, según su propósito.

2.1.3.1. Flexibilidad vs. Seguridad

Desde el punto de vista de diseño, los CPSMs donde predomina la *Flexibilidad*, buscan minimizar la capa de abstracción de la plataforma y especificar las interfaces abstractas de forma tal que aproximen a las APIs de las plataformas subyacentes. De esta manera se intenta eliminar la menor cantidad de opciones presentes en las interfaces nativas como consecuencia del proceso de abstracción.

En el caso de los CPSMs enfocados en la *Seguridad*, se busca organizar las primitivas existentes en las interfaces nativas, en conjuntos funcionales distinguibles con el objeto de incrementar la cohesión y uniformidad de las interfaces[Sch92, p. 6]. Como consecuencia de la reorganización de las interfaces, los CPSMs enfocados en la *Seguridad* suelen presentar conceptos diferentes a los existentes en las interfaces nativas, por ejemplo, objetos que encapsulen la secuencia de invocación de las distintas primitivas reduciendo los errores que puedan cometerse por omisión o utilización de argumentos incorrectos a las primitivas (ver Listado 1.7 y la explicación posterior). Un ejemplo de esto último se presentó en la sección 1.2.3, donde en el Listado 1.6 se utiliza ACE para resolver un ejemplo de implementación de un programa servidor. En la interfaz de *BSD Sockets*, para establecer una conexión (*SOCK_STREAM*) a través de un *socket* activo es necesario invocar en orden *socket()* y luego a *connect()*. Para recibir una conexión en un *socket* pasivo, las primitivas a invocar son *socket()*, *bind()*, *listen()* y *accept()*. Todo el mecanismo podría encapsularse en una única primitiva para cada caso, i.e. *connect()* (o *activeOpen()*) y *listen()* (o *pasiveOpen()*). En el Listado 1.6 es posible apre-

ciar como el CPSM ACE presenta una implementación del patrón *Acceptor-Connector* [Sch97], donde existe una clase separada para cada concepto: *socket* activo (*SOCK_Connector*), *socket* pasivo (*SOCK_Acceptor*) y *socket* de transmisión de datos (*SOCK_Stream*).

Como consecuencia del tipo de capa de abstracción presente en los CPSMs, según su propósito, se desprende que los CPSMs centrados en la *Flexibilidad* generalmente utilizan un paradigma estructurado (debido a que las interfaces nativas que deben aproximar están especificadas en este paradigma³⁷). En contraposición, los CPSMs enfocados en la *Seguridad* suelen utilizar un paradigma orientado a objetos que permita mejorar la abstracción y encapsular las construcciones que aumentan la propensión a cometer errores, en interfaces más simples y seguras.

Entre las ventajas asociadas al predominio de la *Flexibilidad*, i.e. minimizando la capa de abstracción, cabe mencionar el soporte nativo (dependiente de la plataforma) y la máxima expresividad y *Performance*. En cambio, un enfoque centrado en la *Seguridad* presenta la ventaja de disminuir la propensión a cometer errores y la complejidad de suplir funcionalidades inexistentes. A continuación, se desarrollan en detalle estas características.

El soporte nativo consiste en proveer una interfaz lo más similar posible a las interfaces nativas de las plataformas consideradas. El principal impacto de un buen soporte nativo es aprovechar el conocimiento existente de programadores expertos en una plataforma en particular, evitando la necesidad de aprender nuevas construcciones y por lo tanto los errores asociados a la inexperiencia. Formalmente, un buen soporte nativo implica que para un servicio s_m , su interfaz abstracta i_m guarde la mayor similitud posible³⁸ con las interfaces nativas de dicho servicio, i_m^n en la mayor cantidad de las plataformas, p^n de \mathcal{P} :

Soporte nativo para el servicio s_m busca maximizar: $i_m \approx i_m^n, \forall n \in [1, N]$

La búsqueda de la máxima *Flexibilidad* y *Performance* viene asociada a la relación que guardan las interfaces abstractas del CPSM y las interfaces nativas expuestas por las plataformas subyacentes. La reducción de la capa de abstracción reduce el impacto del CPSM sobre la *Performance* debido a que el trabajo del CPSM se transforma en una simple traducción de nombres, formatos y valores. Adicionalmente, un CPSM de propósito específico puede proveer un máximo nivel de *Flexibilidad*, al exponer una interfaz abstracta tan expresiva como lo permitan las plataformas subyacentes.

Por ejemplo, considérese un CPSM donde s_2 represente un servicio de Concurrencia, p^1 sea Microsoft Windows y p^2 sea GNU/Linux. En este caso, un CPSM de *Flexibilidad* predominante podría proveer funciones que permitan crear, bloquear y desbloquear un *Mutex* o un semáforo, mientras que un CPSM centrado en la *Seguridad* podría intentar encapsular algunos detalles de implementación, encapsulando conjuntamente opciones de configuración, como proveer una clase *Lock* sin métodos explícitos para ingresar (bloquear) y abandonar (desbloquear) la sección crítica protegida [Kem06].

³⁷Más aún, generalmente son interfaces en el lenguaje de programación C, como el caso de Microsoft Windows y UNIX. Nótese que el CPSM expondrá unas interfaces abstractas que buscan la mayor similitud posible, sintáctica y semántica, con las interfaces nativas de las plataformas con las que es compatible. En este sentido debe entenderse la “aproximación” de interfaces nativas.

³⁸La similitud entre dos interfaces, representada mediante el operador \approx , se presenta sólo a los fines de ilustrar, en términos del modelo formal, las características del soporte nativo. Los criterios para evaluar la similitud entre dos interfaces dependen de muchos factores, entre ellos el uso que se desee hacer de las interfaces, el propósito del CPSM y el tipo de programas que se desea soportar, etc.

Una clara desventaja asociada a maximizar la *Flexibilidad*, vulnerando la *Seguridad*, es que el usuario del CPSM es responsable por la administración de recursos (como un *file descriptor*, o toda la estructura de memoria y ejecución de un *thread*). Así mismo, es el usuario quien debe garantizar la consistencia del estado inicial, intermedio y final de las estructuras que posee. Volviendo al ejemplo anterior, un error clásico en programación concurrente es la aparición de *deadlocks* debido a asimetría de *locks* y *unlocks* sobre un *Mutex* en una función con muchos puntos de retorno [Sch99a].

El siguiente listado muestra una función que procesa una solicitud de acceso a una base de datos local. En él se puede apreciar la asimetría en la estrategia de bloqueo: a una invocación de *lock* le corresponden tres invocaciones de *unlock*.

```
1 msg_t msg;
2 while (!readMessage(&msg))
3 {
4     lock(&mutex); // ingresar en la sección crítica
5     {
6         attr_t attrs;
7         doc_t doc;
8
9         if (db_getAttribs(&db, msg.doc_id, &attrs))
10        {
11            unlock(&mutex); // abandonar la sección crítica
12            continue;
13        }
14
15        msg.payload.ver = attrs.version;
16
17        if (db_retrieve(&db, msg.doc_id, &doc))
18        {
19            unlock(&mutex); // abandonar la sección crítica
20            continue;
21        }
22        memmove(&msg.payload.data, &doc, sizeof(doc_t));
23    }
24    unlock(&mutex); // abandonar la sección crítica
25 }
```

Listado 2.2: Estrategia asimétrica de bloqueo

Estos problemas pueden evitarse mediante el patrón *scoped-locking* [SSRB00] presente en ACE, Boost, etc. Pero, como ya fue mencionado en la sección 1.2.6, la inexistencia de primitivas de bloqueo explícito, como *lock* y *unlock* disminuyen la *Flexibilidad*.

Por último, la reducción de la capa de abstracción mejora la *Performance*. Sin embargo, esta reducción resta libertad de implementación cuando es necesario soportar plataformas que no se aproximan a la interfaz abstracta para algún servicio. En este caso, una interfaz nativa deficiente a bajo nivel obligaría a suplir en *userspace* la funcionalidad faltante en la plataforma considerada. Esto probablemente degrade la *Performance* en esas plataformas aún cuando la funcionalidad faltante no sea crítica para la implementación del servicio. En el Listado 2.6 (ver sección 2.2.1) se ilustra este caso con un ejemplo donde la opción de *timeout* en una primitiva de sincronización bloqueante presenta dificultades de implementación en SGI IRIX.

En la siguiente tabla se resume la comparación precedente:

Propósito	Parámetros	Paradigma	Características
Específico	<i>Flexibilidad y Performance</i>	Estructurado	Soporte nativo y máxima <i>Flexibilidad y Performance</i>
General	<i>Seguridad, Mantenimiento y Compatibilidad</i>	Orientado a Objetos	Menor propensión a cometer errores, menor complejidad y mayor cohesión en interfaces

2.2. Definición de servicios provistos por el CPSM

La actividad de *Definición de servicios provistos por el CPSM* consiste en determinar el tipo y número de servicios que serán provistos por el CPSM y qué interfaces abstractas se expondrán para cada servicio. Esta actividad es el principal factor determinante de la *Flexibilidad* de un CPSM (ver sección 1.2). De acuerdo con la relación entre los parámetros del CPSM, mencionada en la sección 1.2.6, la *Flexibilidad* y la *Compatibilidad* se limitan mutuamente. En la sección 2.3 se presenta la principal actividad involucrada en la determinación de la *Compatibilidad* del CPSM y que por lo tanto, se encuentra estrechamente relacionada con esta actividad.

El objetivo de la *Definición de servicios provistos por el CPSM*, es la construcción de un listado detallado de servicios a los que el CPSM proveerá acceso de forma independiente de la plataforma.

La *Definición de servicios provistos por el CPSM*, permite un acceso inicial a las características del CPSM, y, en algunos casos, a las limitaciones del mismo³⁹.

En el modelo formal, la definición de los servicios se representa mediante el conjunto \mathcal{S} .

A continuación se describen algunos servicios comunes. Esta descripción es el resultado de un análisis de diversos CPSMs, entre ellos ACE, NSPR, SDL, Boost y wxWidgets⁴⁰:

- **Concurrencia:** este servicio incluye todas las primitivas y recursos necesarios para utilizar concurrencia y sincronización dentro de un único proceso. Entre los recursos (u objetos, dependiendo del paradigma) se puede mencionar *Thread*, *Mutex*, *Semaphore*, *Monitor*, *Condition Variables*, etc[LB96]. No es imprescindible que se provean todos los recursos indicados previamente. Sin embargo, a los fines prácticos, se requiere la existencia del recurso *Thread* y al menos un tipo de *lock*, como por ejemplo *Mutex*. Este servicio se abreviará como “CONC”.
- **Comunicaciones:** este servicio permite tener acceso a primitivas de comunicación, generalmente basadas en *BSD Sockets* o interfaces de alto nivel que encapsulen las mismas. Es posible realizar una transferencia de datos entre procesos dentro de una misma computadora (mismo *host*, mediante la interfaz *loopback*) o en diferentes computadoras conectadas a través de una red. Posibles recursos provistos por este servicio son: *TCP Socket*, *UDP Socket*, etc. Este servicio se abreviará como “COMM”.

³⁹Por ejemplo, considérese un CPSM que provea un servicio de administración de procesos. Si dicho servicio incluye una primitiva similar a la función *fork()* de UNIX, el desarrollo del CPSM presentará dificultades para implementar el servicio en plataformas como Microsoft Windows, que no proveen una operación similar a *fork()*. En este caso, se podría limitar la expresividad de *fork()* y proveer únicamente una versión que inmediatamente luego de *fork()* ejecute *exec()*, ya que en Windows esta funcionalidad se puede obtener mediante la familia de funciones *_spawn()*[Mic08c].

⁴⁰En el Capítulo 4 se detallan las características individuales de estos CPSMs.

- **Filesystem:** este servicio encapsula las particularidades del *filesystem* subyacente, entre ellas se puede mencionar: símbolos reservados en nombres de archivos y directorios, símbolo delimitador de directorios, iteración del contenido de un directorio, operaciones sobre archivos y directorios, etc. Posibles recursos provistos por este servicio son: *File*, *Directory*, etc⁴¹. Este servicio se abreviará como “FILE”.
- **Bibliotecas dinámicas:** este servicio permite asociación explícita de bibliotecas en tiempo de ejecución y la importación de funciones y/o recursos existentes en dichas bibliotecas. Un posible recurso provisto por este servicio es *Library* (que representa a una biblioteca dinámica asociada explícitamente). Este servicio se abreviará como “DL”.
- **Comunicación entre procesos:** este servicio permite la comunicación y sincronización entre procesos que se ejecutan en una misma computadora. Dentro de las primitivas y recursos incluidos en este servicio pueden mencionarse *Semaphore*, *Shared Memory*, *Unnamed Pipe*, *Named Pipe*⁴², *Message Queue*, etc. Este servicio se abreviará como “IPC”.

Esta lista se resume en la siguiente tabla:

Nombre	Abreviatura	Posibles recursos
Concurrencia	“CONC”	<i>Thread, Mutex, Semaphore, Monitor, Condition Variables, etc</i>
Comunicaciones	“COMM”	<i>TCP Socket, UDP Socket, etc</i>
Filesystem	“FILE”	<i>File, Directory, etc</i>
Bibliotecas dinámicas	“DL”	<i>Library</i>
Comunicación entre procesos	“IPC”	<i>Semaphore, Shared Memory, Unnamed Pipe, Named Pipe, Message Queue, etc</i>

A lo largo de este trabajo se utilizará la caracterización precedente como marco de referencia para definir los servicios provistos por un CPSM.

Un aspecto fundamental de la *Definición de servicios provistos por el CPSM* es la especificación de las interfaces de cada servicio.

2.2.1. Especificación de las interfaces de cada servicio provisto por el CPSM

La *Especificación de las interfaces de cada servicio provisto por el CPSM* consiste en definir, para cada servicio, cada una de las operaciones que lo componen. La definición de cada operación debería incluir el nombre de la operación, los parámetros que recibe, el valor de retorno, posibles excepciones, pre y post condiciones, etc. La especificación podría incluir diagramas de interacción y de transición de estado para documentar los conceptos representados, no sólo a través de sus operaciones primitivas, sino también de su estado interno. La actividad

⁴¹Boost provee la funcionalidad descrita como parte del servicio de *Filesystem* (a través de la biblioteca *Boost.Filesystem*). La especificación de *Boost.Filesystem* será incluida en el *C++ Technical Report 2*[Daw08a], por lo que pasará a formar parte del estándar del lenguaje[Daw08a].

⁴²También denominado “FIFO”

de especificación también puede incluir la organización de las operaciones en núcleos funcionales que permitan distinguir los diferentes servicios aumentando, así, la cohesión y uniformidad de las interfaces [Sch92, p. 6].

Existen diversas maneras de definir las interfaces de un servicio, tales como *headers* (.h / .hpp) de una biblioteca o *interfaces* en Java [Ses00].

Merecen especial atención los lenguajes cuyo propósito es definir interfaces. Estos lenguajes denominados IDLs (*Interface Definition Languages*) suelen ser una buena opción para especificar las interfaces de un servicio. En particular, el OMG IDL [Obj], utilizado en CORBA (*Common Object Request Broker Architecture*), o el XPIDL [Moz07b] de Mozilla, basado en el anterior. El RPC Language [Dig96], usado para definir interfaces en RPC puede dar una idea de un lenguaje de definición de interfaces para programación estructurada, si bien ha quedado obsoleto.

Una de las principales ventajas de los IDL es la posibilidad de compilar las especificaciones generando código fuente de diferentes lenguajes de programación. Como consecuencia, las interfaces resultantes se mantienen únicamente a través de su especificación IDL y no es necesario interpretar la especificación para llevarla al lenguaje de programación en el cual se desarrollará el CPSM. En el Listado 2.3 se muestra un ejemplo de una interfaz simplificada de un servicio de Comunicaciones especificada en XPIDL. Luego, en el Listado 2.4 se muestra un fragmento del código C++ generado a partir de la compilación de la interfaz especificada en el Listado 2.3.

```

1 [uuid(89b20299-c4a8-46f4-ab9d-b8164bd39b8d)]
2 interface iSocket
3 {
4     boolean connect(in string host, in unsigned short port);
5     boolean listen(in unsigned short port);
6
7     unsigned long send(in string buffer, in unsigned long size);
8     unsigned long recv(inout string buffer, in unsigned long size);
9 };

```

Listado 2.3: Especificación de la interfaz de un servicio de comunicaciones en XPIDL

```

1 /* ... */
2
3 /* starting interface:    iSocket */
4 #define ISOCKET_IID_STR "89b20299-c4a8-46f4-ab9d-b8164bd39b8d"
5
6 #define ISOCKET_IID \
7     {0x89b20299, 0xc4a8, 0x46f4, \
8     { 0xab, 0x9d, 0xb8, 0x16, 0x4b, 0xd3, 0x9b, 0x8d }}
9
10 class NS_NO_VTABLE iSocket {
11 public:
12
13     NS_DEFINE_STATIC_IID_ACCESSOR(ISOCKET_IID)
14
15     /* boolean connect (in string host, in unsigned short port); */
16     NS_IMETHOD Connect(const char *host, PRUint16 port, PRBool *_retval) = 0;
17
18     /* boolean listen (in unsigned short port); */
19     NS_IMETHOD Listen(PRUint16 port, PRBool *_retval) = 0;
20
21     /* unsigned long send (in string buffer, in unsigned long size); */
22     NS_IMETHOD Send(const char *buffer, PRUint32 size, PRUint32 *_retval) = 0;
23
24     /* unsigned long recv (inout string buffer, in unsigned long size); */
25     NS_IMETHOD Recv(char **buffer, PRUint32 size, PRUint32 *_retval) = 0;

```

```

26
27 };
28
29 /* Use this macro when declaring classes that implement this interface. */
30 #define NS_DECL_ISOCKET \
31   NS_IMETHOD Connect(const char *host, PRUint16 port, PRBool *_retval); \
32   NS_IMETHOD Listen(PRUint16 port, PRBool *_retval); \
33   NS_IMETHOD Send(const char *buffer, PRUint32 size, PRUint32 *_retval); \
34   NS_IMETHOD Recv(char **buffer, PRUint32 size, PRUint32 *_retval);
35
36 /* ... */

```

Listado 2.4: Fragmentos del resultado de compilar la especificación del listado anterior

En el Listado 2.4 es posible apreciar el resultado de compilar una interfaz definida en IDL. La declaración de una clase que implementa la interfaz especificada se puede realizar mediante la expansión de la *macro* `NS_DECL_ISOCKET` [TO03]. XPIDL es el lenguaje de especificación de interfaces que se utiliza en Mozilla Cross-Platform Component Object Model (XPCOM), que es una tecnología sustentada sobre NSPR [Par01b]. Debido a esto, los tipos de los parámetros especificados en XPIDL son traducidos a tipos propios de NSPR (e.g. *unsigned long* se traduce a `PRUint32`, *boolean* a `PRBool`⁴³, etc).

Una alternativa a un IDL consiste en especificar las interfaces directamente en el lenguaje de programación del CPSM. Una ventaja considerable de esta alternativa consiste en simplificar tanto la actividad de especificación cuanto la interpretación de las especificaciones de los servicios. Esto se debe a que un programador que utilice el CPSM no necesitará tener conocimientos de un lenguaje adicional, y potencialmente muy diferente, al del CPSM como podría ser un IDL. Sin embargo, esta alternativa tiene algunas desventajas. Una de ellas es cierta ambigüedad semántica respecto de qué significa cada operación, y en qué consisten los argumentos. Dependiendo del lenguaje de programación en que se desarrolle el CPSM, también puede darse el caso en que no se expliciten las posibles condiciones de error, excepciones, etc.

Una buena solución intermedia es iniciar la especificación utilizando el lenguaje de programación del CPSM, ampliando esta especificación con características semánticas, tanto aquellas propias de IDL cuanto indicaciones típicas de documentación. Entre estas últimas deben incluirse pre y postcondiciones, indicación de valor de retorno y posibles excepciones, tipo de parámetros (*in*, *out*), etc. Una especificación ampliada, realizada en el lenguaje de programación del CPSM, contribuye en varios aspectos a mejorar la calidad de la documentación de las interfaces abstractas del CPSM, entre ellas puede mencionarse el criterio de *Modular understandability*⁴⁴ y el principio de *Self-Documentation*⁴⁵ [Mey97].

El sistema de documentación Doxygen [vH06] define una serie de indicadores que pueden utilizarse para especificar las interfaces, con la ventaja adicional de poder generar la documentación de usuario de la API del CPSM de forma automática. A continuación se presenta una selección de algunos indicadores que podrían utilizarse en la especificación⁴⁶:

- `\param <nombre> {descripción}`: explicita el significado del parámetro indicado, se

⁴³Debe considerarse que NSPR es un CPSM escrito en C donde no existe el tipo de dato *bool* de C++.

⁴⁴*Modular understandability* podría traducirse como *criterio de comprensibilidad* e implica que la especificación de las interfaces, y el funcionamiento de los servicios asociados, puede comprenderse sin necesidad de información externa a la especificación propiamente dicha.

⁴⁵*Self-Documentation* podría traducirse como *principio de autodocumentación* e implica que la especificación de una interfaz forma parte de la interfaz propiamente dicha. Es decir, toda la documentación de una interfaz forma parte de la especificación de dicha interfaz.

⁴⁶En el manual de Doxygen [vH06] se detalla la lista completa de los indicadores con sus variantes y sintaxis.

debería incluir en la descripción el tipo de parámetro: de entrada (*in*), de salida (*out*) o de entrada-salida (*inout*).

- `\return {descripción}`: indica el significado del valor de retorno y cómo deberían interpretarse posibles valores del mismo.
- `\exception <nombre> {descripción}`: declara que la operación puede arrojar la excepción indicada. Se debe especificar el significado con el cual debería interpretarse la excepción.
- `\pre {descripción}`: detalla las precondiciones de la operación.
- `\post {descripción}`: detalla las postcondiciones de la operación.

En el Caso de Estudio se utiliza el método de especificar las interfaces en el lenguaje del CPSM, enriquecido con los indicadores precedentes (ver sección 3.2.1).

La especificación de interfaces para un servicio determina la expresividad de la interfaz abstracta de dicho servicio. Maximizar la expresividad de las interfaces abstractas de un servicio aumenta la *Flexibilidad* del CPSM, ya que aumenta las posibilidades de construcción de software sustentado sobre el CPSM. Sin embargo, una especificación de interfaces excesivamente expresiva provoca, como consecuencia, la disminución de la *Compatibilidad*. Esto se debe principalmente a que, cuanto más expresivas sean las interfaces, mayor será la probabilidad de que algunas opciones no sean provistas por la interfaz nativa del servicio para alguna plataforma. Para superar esta limitación sin modificar la especificación de las interfaces, es necesario realizar una implementación en *userspace* que puede vulnerar la *Performance* en dicha plataforma. En la sección 1.2.6 se presentó un ejemplo donde la opción de *timeout* en primitivas bloqueantes de las interfaces de NSPR requería trabajo adicional para soportar la plataforma Windows NT 3.5.1.

El siguiente fragmento de la especificación del servicio de sincronización de SDL ilustra un caso similar:

```
1 /* SDL_mutex.h */
2
3 /* (...) */
4
5 /* Variant of SDL_SemWait() with a timeout in milliseconds, returns 0 if
6  the wait succeeds, SDL_MUTEX_TIMEDOUT if the wait does not succeed in
7  the allotted time, and -1 on error.
8  On some platforms this function is implemented by looping with a delay
9  of 1 ms, and so should be avoided if possible.
10 */
11 extern DECLSPEC int SDLCALL SDL_SemWaitTimeout(SDL_sem *sem, Uint32 ms);
12
13 /* (...) */
```

Listado 2.5: SDL: Primitiva de bloqueo de semáforos con *timeout*

La primitiva especificada, correspondiente a la operación *wait* con *timeout* de un semáforo, genera problemas de implementación entre las diferentes plataformas del CPSM. En particular la implementación para SGI IRIX se presenta a continuación:

```
1 /* irix/SDL_syssem.c */
2
3 /* (...) */
4
5 int SDL_SemWaitTimeout(SDL_sem *sem, Uint32 timeout)
```

```

6 {
7     int retval;
8
9     if ( ! sem ) {
10        SDL_SetError("Passed a NULL semaphore");
11        return -1;
12    }
13
14    /* Try the easy cases first */
15    if ( timeout == 0 ) {
16        return SDL_SemTryWait(sem);
17    }
18    if ( timeout == SDL_MUTEX_MAXWAIT ) {
19        return SDL_SemWait(sem);
20    }
21
22    /* Ack! We have to busy wait... */
23    timeout += SDL_GetTicks();
24    do {
25        retval = SDL_SemTryWait(sem);
26        if ( retval == 0 ) {
27            break;
28        }
29        SDL_Delay(1);
30    } while ( SDL_GetTicks() < timeout );
31
32    return retval;
33 }
34
35 /* (...) */

```

Listado 2.6: SDL: Implementación ineficiente debido a funcionalidad inexistente

En el Listado 2.6 puede apreciarse cómo la especificación de la interfaz del servicio de sincronización de SDL, en particular en lo referente al uso de semáforos, resulta demasiado ambiciosa en relación a la *Definición de plataformas compatibles con el CPSM* (ver sección 2.3). Debido a la opción de acotar el tiempo de ejecución de una llamada bloqueante a la operación *wait* de un semáforo (i.e. a través de un *timeout*), la *Flexibilidad* del CPSM se incrementa. Sin embargo, una consecuencia de esta especificación es un impacto directo en la *Performance* del CPSM en algunas plataformas, como resultado de una implementación ineficiente. En el Listado 2.6, la implementación de semáforos en SGI IRIX, demuestra cómo, en el caso de utilizar la opción de *timeout*, es necesario recurrir a un *busy wait* para simular el comportamiento deseado, degradando ampliamente la *Performance*. Una alternativa consiste en disminuir la *Compatibilidad*, eliminando a SGI IRIX de la lista de plataformas compatibles con el CPSM. En el caso de SDL se optó por comprometer la *Performance* e indicar en la documentación de la primitiva *SDL_SemWaitTimeout* que su uso no es recomendado debido a los problemas que puede ocasionar en algunas plataformas (ver Listado 2.5).

Parte de la especificación de interfaces incluye seleccionar un mecanismo consistente⁴⁷ de reportar los posibles errores en las primitivas del servicio. Existen varios mecanismos de reporte de errores, entre ellos se cuentan: valor de retorno de las primitivas, variable global de error y excepciones. El mecanismo de reporte de errores depende de las prestaciones del lenguaje de programación del CPSM (e.g. en C++ será posible utilizar excepciones, mientras que en C no hay soporte para este tipo de mecanismo).

⁴⁷Entiéndase por “mecanismo consistente” un mecanismo sintácticamente coherente en todas las interfaces del CPSM. Es decir, un mecanismo que permita representar las condiciones de error de todas las primitivas de los servicios provistos por el CPSM.

La principal desventaja de utilizar el valor de retorno de las primitivas para reportar condiciones de error es la imposibilidad de especificar primitivas que retornen un valor. Como consecuencia, se aumenta la complejidad de las interfaces que deberán especificar parámetros del retorno (i.e. de tipo *out*). Si se combina, en el valor de retorno, condiciones de error y resultado de la primitiva (este es el caso de la primitiva *recv()* de *BSD Sockets*), se compromete la *Seguridad* de la interfaz, ya que es posible malinterpretar una condición de error como un valor válido de retorno (y viceversa). Un refinamiento de este mecanismo consiste en especificar un único valor de error y utilizar una variable global de error para indicar qué error se produjo. En caso de estar disponibles, resulta más conveniente utilizar excepciones para indicar condiciones de error. La principal ventaja de las excepciones es que desacoplan el mecanismo de error del valor de retorno de las primitivas. Adicionalmente, el uso correcto de excepciones aumenta la *Seguridad* de las interfaces fomentando el control de errores y en muchos casos impidiendo ignorar los posibles errores arrojados por las primitivas[Str01].

2.3. Definición de plataformas compatibles con el CPSM

La actividad de *Definición de plataformas compatibles con el CPSM* consiste en determinar el conjunto de plataformas sobre las que se construirá el CPSM. Dado que el objetivo ulterior de un CPSM es aislar a las capas superiores de los detalles de implementación dependientes de la plataforma para un conjunto de servicios, esta actividad caracteriza el grado de *Compatibilidad* que proveerá el CPSM.

En el modelo formal, la definición de plataformas se representa mediante el conjunto \mathcal{P} .

Es importante recordar que existe una restricción inicial sobre las plataformas candidatas a ser seleccionadas en esta actividad. Esta restricción, denominada *condición de existencia del punto de articulación*, fue presentada en la sección 1.1 y se repite a continuación:

Entre todas las plataformas seleccionadas debe existir al menos un servicio en común (i.e. con interfaces compatibles en todas las plataformas consideradas), o punto de articulación, que será el principal sustento del CPSM. En el modelo formal los puntos de articulación se representan mediante el conjunto Σ , y la restricción de existencia de al menos un punto de articulación, a través de la siguiente expresión:

$$\exists s_0 \in \Sigma$$

Al margen de la restricción precedente, cabe destacar que cuanto mayor sea el número de plataformas seleccionadas, mayor *Compatibilidad* tendrá el CPSM. Es decir, un programa sustentado sobre el CPSM tendrá un mayor número de opciones de ejecución, ya que será compatible con cada una de las plataformas que caracterizan al CPSM.

Existe una fuerte interacción entre la *Definición de plataformas compatibles con el CPSM* y la *Definición de servicios provistos por el CPSM* (ver sección 2.2). Puntualmente, al aumentar el número de plataformas con las que el CPSM es compatible, se reducirán los servicios candidatos a ser provistos por el CPSM[Moz06b]. Esto se debe principalmente a que, al considerar un mayor número de plataformas con interfaces distintas, se incrementa la probabilidad de que en alguna plataforma no exista (y no sea factible construir⁴⁸) una implementación de uno de los servicios requeridos.

⁴⁸La determinación de la factibilidad de construcción de una implementación de un servicio en *userspace* dependerá, entre otras cosas, del propósito del CPSM (ver sección 2.1.3) y del contexto en el cual se requiera utilizar el servicio en cuestión, de las limitaciones que presente la o las plataformas en relación a la funcionalidad inexistente, etc.

2.4. Mecanismos de abstracción en un CPSM

En esta sección se analizan las actividades asociadas a la implementación de los mecanismos de abstracción en un CPSM. Inicialmente se realiza una distinción entre implementaciones alternativas e implementaciones incompatibles. Esta distinción permite dividir el análisis de los mecanismos de abstracción, según su aplicación, en mecanismos de *Compilación selectiva* y mecanismos de *Selección de implementaciones alternativas de un servicio*. Por último, se analiza el mecanismo asociado a la *Inicialización y Finalización de servicios*.

2.4.1. Implementaciones alternativas vs. implementaciones incompatibles

Suponiendo que existen al menos dos implementaciones distintas de una misma interfaz, estas implementaciones pueden clasificarse como implementaciones alternativas o implementaciones incompatibles.

Dada una plataforma, implementaciones alternativas de un servicio serán aquellas que, siendo compatibles con dicha plataforma, proveen una mecánica distinta de implementación, como ser *user-threads* y *kernel-threads*. Este caso se presenta en NSPR donde, dependiendo de la plataforma, es posible utilizar un *wrapper* de la implementación nativa de *threads* de la plataforma, una implementación en *userspace* o una implementación combinada de las dos alternativas anteriores[Moz98]. Otro ejemplo son los mecanismos de sincronización en Microsoft Windows, donde un recurso *Mutex* podría implementarse utilizando *CRITICAL_SECTIONS* o *Windows Mutex Objects*. En el Caso de Estudio se discuten ventajas y desventajas de estas implementaciones alternativas (ver sección 3.3.2). En síntesis, utilizar *CRITICAL_SECTIONS* provee mejor *Performance*, pero una implementación basada en *Windows Mutex Objects* facilita la implementación de *Condition Variables*[SP].

Implementaciones incompatibles de un servicio son aquellas para las cuales existe al menos una plataforma (en \mathcal{P}) en la que no pueden ejecutarse ambas implementaciones, por ejemplo *Winsock* y *BSD Sockets* en relación a GNU/Linux y Microsoft Windows, respectivamente⁴⁹.

2.4.2. Compilación selectiva

Al compilar un CPSM en una plataforma específica, se hace evidente uno de los primeros problemas relacionados con el desarrollo de software capaz de ejecutarse en múltiples plataformas incompatibles entre sí: la selección de un mecanismo de abstracción adecuado.

No debe confundirse el problema de la determinación del mecanismo de abstracción con el de la *Selección de paradigma* (ver sección 2.1.1). Por mecanismo de abstracción, debe entenderse el proceso que se utilizará para lograr la separación efectiva entre las interfaces abstractas que definen el punto de acceso al CPSM (ver sección 2.2.1) y las implementaciones dependientes de la plataforma.

Se denomina “plataforma *target*” a la plataforma donde se ejecutará el CPSM compilado. Normalmente la plataforma *target* coincide con la plataforma subyacente en el entorno de com-

⁴⁹En este punto se excluye la ejecución de programas a través de la emulación de la plataforma (e.g. Wine, Cywin, etc). Si bien es posible ejecutar programas emulando la plataforma, el programa en cuestión no se encuentra en una representación binaria compatible con la plataforma de ejecución. Al utilizar un CPSM, la representación binaria del programa sustentado sobre él resulta compatible con la plataforma y es posible ejecutar dicho programa sin necesidad de un entorno de emulación. En las afirmaciones anteriores debe entenderse “representación binaria de un programa” al resultado de compilar el código fuente de un programa en un ejecutable o biblioteca binaria.

pilación⁵⁰. Un caso especial, donde la plataforma *target* (i.e. aquella en la cual se ejecutará el código compilado) no es la plataforma la plataforma de compilación (i.e. plataforma *host*), consiste en utilizar técnicas de *cross-compiling* para compilar en una plataforma, ejecutables destinados a otra plataforma[Min04]. En lo sucesivo se utilizará el término “plataforma *target*” para hacer referencia a la plataforma donde se ejecutará el código compilado, contemplando, de esta manera, el caso donde plataforma de ejecución no coincide con la plataforma de compilación⁵¹.

Por compilación selectiva debe entenderse la posibilidad de compilar sólo una porción del código del CPSM, en particular, sólo la porción compatible con la plataforma *target*. Para cada servicio disponible en el CPSM sólo deberán compilarse las implementaciones compatibles con la plataforma *target*⁵² (i.e. se deben excluir las implementaciones incompatibles con dicha plataforma, ver sección 2.4.1).

Si bien el problema de la compilación selectiva se aplica tanto a implementaciones alternativas cuanto a incompatibles, la selección del primer tipo de implementaciones tiene menores restricciones que la del segundo. Esto se debe a que en el caso de implementaciones alternativas, es posible compilar todas las implementaciones alternativas disponibles y luego utilizar un patrón de diseño relevante⁵³ para decidir (potencialmente en tiempo de ejecución) cuál de las implementaciones alternativas utilizar. Este caso se analiza en la sección 2.4.3. En esta sección se presentan algunas soluciones al problema de la compilación selectiva, aplicadas a la selección de implementaciones compatibles con la plataforma *target*.

Existen varios métodos para resolver el problema de la compilación selectiva, a continuación se presentan dos de los más utilizados.

2.4.2.1. Compilación condicional

En lenguajes de programación con directivas condicionales de preprocesamiento, como C, C++ y Pascal, es posible explotar las ventajas de la compilación en dos etapas, preprocesamiento y compilación propiamente dicha, para seleccionar en forma condicional qué porción de código compilar.

El método de compilación condicional implica, en primer lugar, la asignación de una o más variables que permitan distinguir la plataforma *target*, y luego utilizar las directivas de compilación condicional del preprocesador para evaluar el valor de estas variables y determinar qué porción de código compilar.

La compilación condicional permite resolver el problema de la compilación selectiva, embebiendo directivas de preprocesador en el cuerpo de las funciones o métodos que implementan las primitivas de cada interfaz (e.g. *#ifdef*, *#else*, *#endif*)[Iqb]. Sin embargo, el código resultante será muy difícil de leer, comprender y mantener[Sch99b]. Adicionalmente, este tipo de solución deja muy poco lugar para optimizaciones dependientes de la plataforma, debido a que éstas se deberán hacer dentro de cada bloque de compilación condicional. En la sección 1.2.3 se presentan dos ejemplos de compilación condicional en C/C++ (ver Listados 1.4 y 1.5).

⁵⁰i.e. la distribución binaria, dependiente de la plataforma, del CPSM (ejecutables, bibliotecas, etc) se compila en una plataforma con las características de la plataforma *target*. En particular, en la misma plataforma *target*.

⁵¹Las técnicas de *cross-compiling*, y los problemas asociados a estas técnicas, escapan al alcance de esta tesis, sin embargo más información puede encontrarse en: <http://www.libsdl.org/extras/win32/cross/README.txt>, <http://gcc.gnu.org/install/specific.html> y <http://www.mingw.org/mingwfaq.shtml#faqcross>.

⁵²Nótese que, para que la plataforma considerada sea compatible con el CPSM se requiere que exista (o sea posible construir) al menos una implementación para cada interfaz de cada servicio provisto.

⁵³e.g. *Abstract Factory*, *Factory Method*, etc.

El siguiente ejemplo ilustra una aplicación de compilación condicional en ACE:

```
1 /* (...) */
2
3 int
4 ACE_OS::mutex_lock (ACE_mutex_t *m,
5                    const ACE_Time_Value &timeout)
6 {
7 #if defined (ACE_HAS_THREADS) && defined (ACE_HAS_MUTEX_TIMEOUTS)
8
9 # if defined (ACE_HAS_PTHREADS)
10
11 /* (...) */
12
13 # elif defined (ACE_HAS_WTHREADS)
14 // Note that we must convert between absolute time (which is passed
15 // as a parameter) and relative time (which is what the system call
16 // expects).
17
18 /* (...) */
19
20 # elif defined (ACE_VXWORKS)
21
22 // Note that we must convert between absolute time (which is passed
23 // as a parameter) and relative time (which is what the system call
24 // expects).
25
26 /* (...) */
27
28 # endif /* ACE_HAS_PTHREADS */
29
30 #else
31 ACE_UNUSED_ARG (m);
32 ACE_UNUSED_ARG (timeout);
33 ACE_NOTSUP_RETURN (-1);
34 #endif /* ACE_HAS_THREADS && ACE_HAS_MUTEX_TIMEOUTS */
35 }
36
37 /* (...) */
```

Listado 2.7: Fragmento de la implementación de la primitiva *lock()* de *Mutex* en ACE.

El Listado 2.7 muestra un fragmento de la implementación de la primitiva *lock()* de un *Mutex* en ACE. En dicho listado es posible apreciar la complejidad asociada al mecanismo de compilación condicional, aplicado a la abstracción de múltiples plataformas.

La compilación condicional puede ser útil en casos excepcionales, donde el código que debe abstraerse es muy pequeño. Cuando el proceso de abstracción de las interfaces nativas, para exponer una única interfaz abstracta, sea más complejo, el método de compilación condicional presenta grandes desventajas, la principal de ellas es la complejidad de *Mantenimiento* (ver sección 1.2.3). En estos casos es posible obtener mejores resultados mediante el método de separación física presentado a continuación.

2.4.2.2. Separación física

El método de compilación selectiva a través de separación física de archivos y/o directorios supone diferir la selección de las implementaciones compatibles con la plataforma *target* al momento de compilación, en lugar de preestablecerla por código como propone el método anterior. Por separación física se entiende utilizar diferentes archivos (potencialmente en diferentes directorios) para cada implementación. Luego, en tiempo de compilación determinar, en función

de la plataforma *target*, qué directorio compilar⁵⁴.

Dependiendo del sistema de compilación que se utilice, se tendrán distintas prestaciones para realizar la selección. Por ejemplo, en el caso de utilizar *GNU make*, es posible determinar el directorio de compilación según el “objetivo de compilación” indicado por parámetro, utilizar la opción *-I* para seleccionar un directorio de inclusión de *Makefiles* o , en caso de estar presente, utilizar la variable de entorno *OSTYPE*⁵⁵[RMSS06].

El siguiente ejemplo ilustra el método de compilación selectiva a través de separación física de archivos y directorios para un servicio sencillo de concurrencia en sistemas POSIX y Microsoft Windows. La compilación se realiza mediante el programa *GNU make*, que está incluido en todas las distribuciones importantes de GNU/Linux. En Microsoft Windows se utiliza el entorno de compilación MSYS⁵⁶ que incluye una versión de *GNU make* para Microsoft Windows.

La estructura de directorios del ejemplo se presenta en la Figura 2.4.

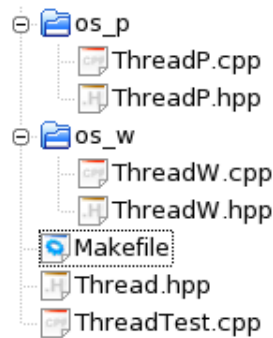


Figura 2.4: Estructura de directorios

El ejemplo consta de los siguientes archivos:

- *Makefile*: archivo que define el proceso de compilación (ver Listado 2.8).
- *Thread.hpp*: definición del servicio de Concurrencia (ver Listado 2.9).
- *os_p/ThreadP.hpp*: interfaz del servicio definido en *Thread.hpp* para sistemas POSIX basada en la biblioteca POSIX Threads[IEE04] (ver Listado 2.10).
- *os_p/ThreadP.cpp*: implementación del servicio definido en *os_p/ThreadP.hpp* (ver Listado 2.11).
- *os_w/ThreadW.hpp*: interfaz del servicio definido en *Thread.hpp* para Microsoft Windows (ver Listado 2.12).
- *os_w/ThreadW.cpp*: implementación del servicio definido en *os_p/ThreadW.hpp* (ver Listado 2.13).

⁵⁴La selección de implementaciones alternativas (ver sección 2.4.2) puede resolverse, en tiempo de compilación, mediante opciones al sistema de compilación[Moz98].

⁵⁵e.g. en GNU/Linux, Cygwin y MinGW/MSYS en Microsoft Windows.

⁵⁶MSYS puede descargarse gratuitamente de <http://www.mingw.org/msys.shtml> y es el entorno de compilación utilizado por Mozilla Foundation para compilar sus productos en Microsoft Windows[Moz08b].

- *ThreadTest.cpp*: programa de prueba que permite evaluar el servicio definido en *Thread.hpp* (ver Listado 2.14).

A continuación se presenta el contenido de los archivos que componen el ejemplo:

- *Makefile*:

```

1 ifeq ($(OSTYPE),msys)
2     WINDOWS := 1
3 endif
4
5 ifdef WINDOWS
6     CXX      :=cl.exe
7     CC       :=cl.exe
8     CXX_OUT  :=-Fo'$(1)\'
9     CXXFLAGS := -nologo -MD -W3 -GX -O2                \
10                -D 'NDEBUG' -D 'WIN32' -D '_WINDOWS' -D '_MBCS' \
11                -D '_USRDLL' -D 'MSVC' -FD -c          \
12                -nologo
13     LD       :=link.exe
14     LD_OUT   :=-out:$(1)
15     LDFLAGS  := kernel32.lib user32.lib gdi32.lib winspool.lib \
16                comdlg32.lib advapi32.lib shell32.lib ole32.lib \
17                oleaut32.lib uuid.lib odbcc32.lib odbccp32.lib -nologo \
18                -pdb:none -machine:I386
19
20     OBJ_SUFFIX :=obj
21
22     EXE_SUFFIX :=.exe
23 else
24     CXX      :=g++
25     CC       :=gcc
26     CXX_OUT  :=-o $(1)
27     CXXFLAGS :=-pedantic -Wall -Wno-long-long
28
29     LD       :=g++
30     LD_OUT   :=-o $(1)
31     LDFLAGS  := $(LDFLAGS) $(CXXFLAGS) -pthread -ldl
32
33     OBJ_SUFFIX :=o
34 endif
35
36 ifdef WINDOWS
37 OS_DIR :=os_w
38 else
39 OS_DIR :=os_p
40 endif
41
42 SRCS :=$(wildcard $(OS_DIR)/*.cpp)
43 OBJS :=$(SRCS:.cpp=$(OBJ_SUFFIX))
44
45 %.$(OBJ_SUFFIX) : %.cpp
46     $(CXX) $(CXXFLAGS) -c $< $(call CXX_OUT,$@)
47
48 thread_test$(EXE_SUFFIX) : ThreadTest$(OBJ_SUFFIX) $(OBJS)
49     $(LD) $(LDFLAGS) $~ $(call LD_OUT,$@)
50
51 .PHONY: clean
52 clean:
53     rm -f *.$(OBJ_SUFFIX) $(OBJS) thread_test$(EXE_SUFFIX) *.idb
54
55 .PHONY: test
56 test: thread_test$(EXE_SUFFIX)
57     ./thread_test$(EXE_SUFFIX)

```

Listado 2.8: Makefile que provee abstracción por separación física de archivos y directorios

Nótese que el primer bloque condicional del *Makefile* presentado en el Listado 2.8 encapsula las diferencias entre las plataformas y las opciones esperadas por el compilador de cada plataforma. En GNU/Linux se utiliza GNU g++ (definido en la línea 24), mientras que Microsoft Windows se utiliza Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8804 for 80x86 (definido en la línea 6).

■ *Thread.hpp*:

```

1 #ifndef _THREAD_HPP__
2 #define _THREAD_HPP__
3
4 #include <memory>
5
6 class Thread
7 {
8     public:
9         typedef void (*func_t)(void* arg); ///< función principal del Thread
10        typedef std::auto_ptr<Thread> Thread_t; ///< implementación de Thread
11
12        Thread() {}
13        virtual ~Thread() {}
14
15        /** Ejecutar ‘f’ en otro thread con ‘arg’ como argumento.
16         * \param f [in] Función a ejecutar en un nuevo thread.
17         * \param arg [in] Argumento enviado a la función ‘f’.
18         * \return true si la operación fue satisfactoria, false en caso
19         *         contrario.
20         */
21        virtual bool start(func_t f, void* arg) = 0;
22
23        /** Esperar a que finalice la ejecución del thread iniciado mediante
24         * start().
25         */
26        virtual void join() = 0;
27
28        /** Obtener una nueva instancia de Thread.
29         * \return un objeto Thread_t que es un auto_ptr a una implementación
30         * de Thread en la plataforma subyacente.
31         */
32        static Thread_t create();
33 };
34
35 #endif

```

Listado 2.9: Definición de un servicio de Concurrencia mínimo

Básicamente se definen tres operaciones:

- *Thread::create()* permite obtener una implementación adecuada del servicio de definido en *Thread.hpp* para la plataforma subyacente. Esta función podría reemplazarse por una implementación del patrón *Abstract Factory*[GHJV95] que permita seleccionar implementaciones alternativas (en caso de existir) para cada plataforma.
- *start()* inicia la ejecución de la función indicada en “f” en un nuevo thread.
- *join()* espera que el thread finalice su ejecución.

■ *os_p/ThreadP.hpp*:

```

1 #ifndef _THREADP_HPP__
2 #define _THREADP_HPP__
3
4 #include "../Thread.hpp"

```

```

5
6 #include <pthread.h>
7
8 class ThreadP : public Thread
9 {
10     public:
11         ThreadP();
12         virtual ~ThreadP();
13
14         bool start(func_t f, void* arg);
15         void join();
16
17     protected:
18         pthread_t _handle;
19         func_t _f;
20         void* _arg;
21
22         static void* _thread_start(void* arg);
23 };
24
25 #endif

```

Listado 2.10: Interfaz del servicio para sistemas POSIX

La función `ThreadP::thread_start()` permite compatibilizar el prototipo de la función principal del thread definida en `Thread.hpp` (`Thread::func_t`, ver Listado 2.9) con el prototipo esperado por la función `pthread_create()`.

■ `os_p/ThreadP.cpp`:

```

1 #include "ThreadP.hpp"
2
3 Thread::Thread_t Thread::create() { return Thread_t(new ThreadP); }
4
5 ThreadP::ThreadP() : _handle(0), _f(NULL) {}
6
7 ThreadP::~~ThreadP()
8 {
9     join();
10 }
11
12 bool ThreadP::start(func_t f, void* arg)
13 {
14     _f = f;
15     _arg = arg;
16     return (pthread_create(&_handle, NULL, _thread_start, this) == 0);
17 }
18
19 void ThreadP::join()
20 {
21     if (!_handle)
22         return;
23     pthread_join(_handle, NULL);
24     _handle = 0;
25 }
26
27 void* ThreadP::_thread_start(void* arg)
28 {
29     ThreadP* instance = reinterpret_cast<ThreadP*>(arg);
30     if (!instance)
31         return NULL;
32
33     (*instance->_f)(instance->_arg);
34
35     return NULL;

```


36 }

Listado 2.11: Implementación del servicio para sistemas POSIX■ *os_w/ThreadW.hpp*:

```

1 #ifndef _THREADW_HPP_
2 #define _THREADW_HPP_
3
4 #include "../Thread.hpp"
5
6 #include <windows.h>
7
8 class ThreadW : public Thread
9 {
10     public:
11         ThreadW();
12         virtual ~ThreadW();
13
14         bool start(func_t f, void* arg);
15         void join();
16
17     protected:
18         HANDLE _handle;
19         func_t _f;
20         void* _arg;
21
22         static unsigned __stdcall _thread_start(void* arg);
23 };
24
25 #endif

```

Listado 2.12: Interfaz del servicio para Microsoft Windows

La función *ThreadW::thread_start()* permite compatibilizar el prototipo de la función principal del thread definida en *Thread.hpp* (*Thread::func_t*, ver Listado 2.9) con el prototipo esperado por la función *_beginthreadex()*.

■ *os_w/ThreadW.cpp*:

```

1 #include "ThreadW.hpp"
2
3 #include <process.h>
4
5 Thread::Thread_t Thread::create() { return Thread_t(new ThreadW); }
6
7 ThreadW::ThreadW() : _handle((HANDLE)-1L), _f(NULL) {}
8
9 ThreadW::~ThreadW()
10 {
11     join();
12 }
13
14 bool ThreadW::start(func_t f, void* arg)
15 {
16     _f = f;
17     _arg = arg;
18     _handle = (HANDLE)_beginthreadex(NULL,0,_thread_start,this,0,NULL);
19     return (_handle != (HANDLE)-1L);
20 }
21
22 void ThreadW::join()
23 {
24     if (_handle == (HANDLE)-1L)

```

```
25         return;
26
27     WaitForSingleObject( _handle, INFINITE );
28     _handle = (HANDLE)-1L;
29 }
30
31 unsigned ThreadW::_thread_start(void* arg)
32 {
33     ThreadW* instance = reinterpret_cast<ThreadW*>(arg);
34     if (!instance)
35         return NULL;
36
37     (*instance->_f)(instance->_arg);
38
39     return NULL;
40 }
```

Listado 2.13: Implementación del servicio para Microsoft Windows

■ *ThreadTest.cpp:*

```
1 #include "Thread.hpp"
2 #include <stdio.h>
3
4 void test(void* arg)
5 {
6     char* s = reinterpret_cast<char*>(arg);
7     if (!s)
8         return;
9     printf("[thread] start \"%s\" (ENTER)\n", s);
10    getchar();
11    printf("[thread] end \"%s\"\n", s);
12 }
13
14 int main()
15 {
16     // No utilizar buffers en stdout (para evitar tener que hacer fflush des-
17     // pues de cada printf).
18     setvbuf(stdout, NULL, _IONBF, 0);
19
20     char* msg = "hello world";
21
22     // instanciar un nuevo thread
23     Thread::Thread_t t = Thread::create();
24
25     // iniciar el thread
26     printf("[main] start thread\n");
27     t->start(test, msg);
28
29     // esperar a que el thread termine
30     printf("[main] join thread\n");
31     t->join();
32
33     printf("[main] end\n");
34     return 0;
35 }
```

Listado 2.14: Programa de ejemplo

El programa de ejemplo ejecuta una función sencilla (*test()*) en un thread secundario mientras que el thread principal de la aplicación espera. Una vez finalizada la ejecución de dicha función, el thread principal emite un mensaje y el programa termina. La función *setvbuf()* se utiliza para eliminar el *buffer* asociado al *stream* de salida (*stdout*), de esta

forma se evita la necesidad de ejecutar `fflush(stdout)` luego de cada `printf()` para ver el progreso del programa inmediatamente.

Nótese que en el ejemplo precedente se utiliza una versión estática del patrón *Factory Method*[GHJV95] para construir instancias dependientes de la plataforma del recurso *Thread*. La estructura presentada en el ejemplo permite visualizar el mecanismo de separación física de archivos y/o directorios, sin embargo, es posible optimizarla para mejorar la *Performance*. Puntualmente, debe notarse que la interfaz de la clase *Thread* declara las operaciones como métodos abstractos (virtuales puros). De la naturaleza virtual de dichas operaciones, se desprende que cada invocación a una operación requiere un acceso a la tabla de métodos virtuales de la instancia. Este *overhead* es evitable, ya que, en el ejemplo, las clases *ThreadP* y *ThreadW* representan implementaciones incompatibles⁵⁷. Por lo tanto, al compilar en una plataforma determinada, sólo se deberá compilar una implementación de *Thread*. Entonces, el ejemplo podría optimizarse en dos pasos:

- La clase *Thread* no debería ser abstracta (es más, no debería ser polimórfica, i.e. no debería tener métodos virtuales).
- En cada plataforma deberían implementarse directamente las operaciones de la clase *Thread* (y no de una subclase de *Thread*, como en el ejemplo).

En el Caso de Estudio presentado en el Capítulo 3 se utiliza el procedimiento de optimización aquí descrito para reducir el impacto del CPSM sobre la *Performance* de los servicios que provee.

2.4.3. Selección de implementaciones alternativas de un servicio

En algunos casos, un CPSM puede proveer más de una implementación de un servicio en una plataforma dada. En este caso, al realizar un desarrollo sustentado sobre dicho CPSM se podría optar por una de las implementaciones alternativas disponibles en tiempo de compilación, o incluso, se podría seleccionar la implementación en tiempo de ejecución⁵⁸.

Un posible ejemplo de este escenario es:

Un CPSM provee un servicio de Comunicación entre procesos (IPC) cuya interfaz especifica un mecanismo de sincronización a través de *Mutex*. El CPSM es compatible con Microsoft Windows y GNU/Linux. La única implementación de *Mutex* en Windows es una abstracción *Named Mutex Objects* de la API de Windows[Mic08a]. En Linux, provee dos implementaciones de *Mutex*, una basada en *IPC Semaphores*[IEE04] y otra basada en *Fast Userspace Mutex*[Dre08] (*Futex*). Dado que el *kernel* de Linux implementa *Futex* a partir de la versión 2.5, optar por la implementación de IPC podría aumentar la *Compatibilidad*.

Para que resulte posible la selección de una de las implementaciones alternativas de un servicio disponible en una plataforma dada, el CPSM debe proveer un mecanismo de acceso a las

⁵⁷Es decir, no es posible compilar y ejecutar ambas implementaciones en la misma plataforma. Ver sección 2.4.1.

⁵⁸No debe confundirse el tiempo de compilación de un programa sustentado sobre el CPSM con el tiempo de compilación del CPSM. Generalmente los CPSMs se distribuyen en forma de bibliotecas compiladas, que son asociadas a los programas por el *linker*. En esta sección la expresión “tiempo de compilación” se utiliza para hacer referencia al tiempo de compilación de un programa sustentado sobre el CPSM.

diversas implementaciones disponibles sin comprometer la abstracción de la plataforma subyacente. Algunos patrones de diseño facilitan esta tarea, por ejemplo *Wrapper Facade*[Sch99b], *Abstract Factory*[GHJV95], *Factory Method*[GHJV95] y *Component Configurator*[SSRB00].

A continuación se presenta una aplicación del patrón *Factory Method* al escenario precedente:

```

1 #ifndef _MUTEX_HPP__
2 #define _MUTEX_HPP__
3
4 #include <memory>
5
6 class Mutex
7 {
8     public:
9         typedef std::auto_ptr<Mutex> Mutex_t; ///< implementación de Mutex
10        typedef enum
11        {
12            NORMAL,
13            FUTEX
14        } type_t;
15
16        /** Construir un mutex con el nombre dado.
17         * \param name nombre del mutex.
18         */
19        Mutex(const char* name) {}
20        virtual ~Mutex() {}
21
22        /** [bloqueante] Bloquear el mutex (entrar en la sección crítica).
23         * \return true si la operación fue satisfactoria, false en caso
24         *         contrario.
25         */
26        virtual bool lock() = 0;
27
28        /** Liberar el mutex (salir de la sección crítica).
29         * \return true si la operación fue satisfactoria, false en caso
30         *         contrario.
31         */
32        virtual bool unlock() = 0;
33
34        /** Obtener una nueva instancia de Mutex.
35         * \param name nombre del mutex.
36         * \return un objeto Mutex_t que es un auto_ptr a una implementación
37         *         de Mutex en la plataforma subyacente.
38         */
39        static Mutex_t create(const char* name, type_t type = NORMAL);
40 };
41
42 #endif

```

Listado 2.15: Especificación de una interfaz de sincronización *Mutex* con implementaciones alternativas

En el Listado 2.15 puede observarse que el método estático *Mutex::create()* permite seleccionar una de las implementaciones alternativas disponibles. Al implementar este método en cada plataforma se deberá evaluar si el valor especificado es una implementación válida y en caso contrario proveer una implementación *default*. Por ejemplo:

```

1 #include "Mutex.hpp"
2
3 #ifdef WINDOWS // ===== Windows ===
4
5 class NamedMutex; // Implementación concreta
6
7 Mutex::Mutex_t Mutex::create(const char* name, type_t type)
8 {

```

```

9     return Mutex_t(new NamedMutex(name));
10 }
11 /* ... */
12
13 #else // ===== Linux ===
14
15 class IPCMutex; // Implementación concreta
16 class FutexMutex; // Implementación concreta
17
18 Mutex::Mutex_t Mutex::create(const char* name, type_t type)
19 {
20 #ifdef HAS_FUTEX
21     return Mutex_t((type == FUTEX) ? new FutexMutex(name) : new IPCMutex(name));
22 #else
23     return Mutex_t(new IPCMutex(name));
24 #endif // HAS_FUTEX
25 }
26 /* ... */
27
28 #endif // =====

```

Listado 2.16: Selección de implementaciones alternativas de *Mutex* a través del patrón *Factory Method*

El Listado 2.16 muestra la implementación del código dependiente de la plataforma para Microsoft Windows y GNU/Linux. Se utilizan dos *macros* para determinar la plataforma y la versión del kernel disponible, cuando corresponda. Estas *macros* son `WINDOWS` y `HAS_FUTEX` respectivamente. La primera podría determinarse mediante un esquema de compilación similar al presentado en el Listado 2.8, mientras que la segunda podría reemplazarse por la expresión `(LINUX_VERSION_CODE >= KERNEL_VERSION(2,5,0))`⁵⁹. En caso de utilizarse en Windows, el CPSM provee una única implementación de *Mutex*, i.e. *NamedMutex*. En Linux, dependiendo de la versión del *Kernel*, existirá una o dos implementaciones, i.e. *IPCMutex* y *FutexMutex* (únicamente si la versión del *Kernel* es mayor o igual a 2.5).

Un método similar al utilizado en el ejemplo precedente podría aplicarse al Caso de Estudio (ver sección 3.3.2) para proveer implementaciones alternativas, en Microsoft Windows, de *Mutex* (basadas en *CRITICAL_SECTIONS* o en *Windows Mutex Objects*) y *Condition Variables* (basadas en las implementaciones propuestas en [SP]).

Patrones avanzados, como el *Component Configurator*[SSRB00], permiten construir interfaces más configurables, donde las implementaciones alternativas pueden registrarse en tiempo de ejecución. Esencialmente, el patrón *Component Configurator* permite enlazar bibliotecas dinámicamente e invocar las funciones definidas en las mismas. En particular, las bibliotecas exportan una función de inicialización donde registran las implementaciones que proveen. Luego una *Abstract Factory*[GHJV95] actualizada dinámicamente permite obtener las implementaciones alternativas de todas las bibliotecas cargadas. Una explicación más detallada de este patrón puede encontrarse en [SSRB00, Chapter 2: Service Access and Configuration Patterns] y en [Sch93, p. 11].

2.4.4. Inicialización y Finalización de servicios

Un problema asociado a la implementación de servicios de un CPSM consiste en la asimetría de inicialización y finalización de servicios de las diversas implementaciones dependientes de la plataforma. El problema se desprende de las diferencias semánticas⁶⁰ presentes en las interfaces

⁵⁹Estas *macros* están definidas en el archivo `linux/version.h` en los *headers* del *Kernel* de Linux.

⁶⁰Entiéndase por “diferencias semánticas” a aquellas diferencias en la manera de utilizar el servicio a través de su interfaz, independientemente de las diferencias sintácticas de los argumentos de las primitivas (e.g. tipo y

nativas de un servicio. Puntualmente, en lo que respecta a la inicialización y finalización de un servicio, existen dos alternativas: utilizar un mecanismo explícito o uno implícito. El mecanismo explícito consiste en que el CPSM provea primitivas de inicialización y finalización, que deberán ser invocadas antes de la primera, y luego de la última invocación a una primitiva del servicio, respectivamente. El mecanismo implícito consiste en comprobar, en cada primitiva del servicio, si dicho servicio ha sido inicializado previamente y en caso negativo, inicializar el servicio antes de continuar.

Un ejemplo de asimetría en la implementación de las primitivas de inicialización/finalización es el servicio de Comunicaciones en Microsoft Windows (*Winsock*) y en sistemas POSIX (basados en *BSD Sockets*). En Microsoft Windows, el servicio de comunicaciones debe ser inicializado explícitamente (mediante la función *WSAStartup()*) antes de poder utilizar las primitivas de la interfaz [Mic08f]. En cambio, en los sistemas POSIX, la inicialización es implícita y por lo tanto, es posible invocar las primitivas directamente, sin necesidad de ejecutar una función de inicialización.

La asimetría ilustrada en el ejemplo anterior presenta un problema para el desarrollo de un CPSM, debido a que se debe decidir qué tipo de inicialización utilizar para el servicio, y esta decisión impacta en diversos aspectos del CPSM. A continuación se describen algunos de ellos.

Una inicialización explícita puede vulnerar la *Seguridad* del CPSM debido a que los programadores acostumbrados a utilizar una interfaz nativa con inicialización implícita, al migrar al CPSM, probablemente olviden invocar la inicialización explícita del servicio. En el caso de migrar un programa que utiliza *sockets* desde un sistema POSIX (e.g. GNU/Linux) a Microsoft Windows, es muy frecuente omitir la invocación a *WSAStartup()* [Min07]. ACE implementa un esquema de inicialización explícita para el servicio de comunicaciones, mediante la función *ACE_OS::socket_init()*.

La implementación de una inicialización implícita tiene un importante impacto en la *Performance* y la complejidad del código fuente del CPSM, debido a que, cada vez que se invoca una primitiva del servicio, se deberá comprobar que el servicio haya sido inicializado correctamente y en caso de no ser así, se deberá inicializar el servicio antes de continuar. NSPR implementa un esquema de inicialización implícita para el servicio de comunicaciones mediante una combinación de la variable global *_pr_initialized* y la función *_PR_ImplicitInitialization()*. A continuación se muestra el procedimiento mediante el cual se puede calcular la cantidad de invocaciones a *_PR_ImplicitInitialization()* en el código fuente de NSPR. Este número da una idea de la complejidad del código necesario para implementar el esquema de inicialización implícita.

```
prueba@nihil:~/nspr-4.7/mozilla/nsprpub$ grep -r _PR_ImplicitInitialization * | wc -l
102
prueba@nihil:~/nspr-4.7/mozilla/nsprpub$
```

Figura 2.5: Cantidad de invocaciones a *_PR_ImplicitInitialization()* en el código fuente de NSPR

En la Figura 2.5 puede apreciarse que el código fuente de NSPR presenta 102 invocaciones a la función de inicialización implícita⁶¹. Esto implica, entre otras cosas, la existencia de aproximadamente 102 comparaciones para determinar si se ha inicializado previamente el servicio,

número de argumentos). Un ejemplo de diferencias semánticas en una interfaz es la función de C *fclose()* para sistemas POSIX y Microsoft Windows. En ambos casos el prototipo de la función (sintaxis) es idéntico, sin embargo, en la primera plataforma, invocar *fclose* sobre un *stream (FILE*)* previamente cerrado resulta en una señal de *Abort (SIGABRT)*, mientras que en la segunda la función simplemente devuelve un valor de error.

⁶¹En rigor, se debería considerar dentro de la cantidad obtenida, la declaración y la definición de la fun-

y en consecuencia, la complejidad asociada con el *Mantenimiento* del CPSM (i.e. no olvidar la comprobación en la implementación de ninguna primitiva). Nótese que es posible realizar la comprobación dentro de la función de inicialización. Sin embargo, luego de la primer invocación, que efectivamente realiza la inicialización, todas las invocaciones subsiguientes retornarán prematuramente luego de verificar que el servicio ya se encuentra inicializado. Este es un claro ejemplo donde se contraponen *Performance* y *Mantenimiento*, ya que si se opta por realizar primero la comprobación y luego la invocación, es necesario efectuar dicha comprobación en cada primitiva del servicio. Por otro lado, si la comprobación se realiza dentro de la función, se degrada *Performance* debido al *overhead* derivado de la construcción del *stack frame* de la función únicamente para efectuar la comprobación.

Existe un mecanismo que permite implementar inicialización y finalización implícitas sin necesidad de comprobar la inicialización en cada invocación de una primitiva. Este método consiste en utilizar una instancia estática de una clase, que ejecute las rutinas de inicialización en su constructor y las de finalización en su destructor.

```

1 class Initializer
2 {
3     private:
4         Initializer() { /* inicializar servicio */ }
5         static Initializer _the_only_instance;
6     public:
7         ~Initializer() { /* finalizar servicio */ }
8 };
9
10 Initializer Initializer::_the_only_instance;

```

Listado 2.17: Clase de inicialización estática implícita

En el Listado 2.17 se muestra un ejemplo de inicialización implícita mediante una instancia estática. Este mecanismo permite evitar las comprobaciones de inicialización, sin embargo, posee dos desventajas importantes. La primera consiste en la imposibilidad de parametrizar la inicialización, lo que dificulta la configuración del servicio⁶². La segunda desventaja es más grave: puesto que la inicialización tiene lugar en el momento en el que se construye el objeto estático (i.e. *Initializer*), no es posible construir objetos estáticos del CPSM cuya construcción requiera que el servicio se encuentre inicializado. Puntualmente, cualquier objeto estático compite con el objeto de inicialización en tiempo de construcción, ya que el orden de creación de los objetos estáticos dependerá del compilador utilizado⁶³.

Por ejemplo, supóngase la implementación de un servicio de Comunicaciones (con un recurso *Socket*) en Microsoft Windows. Dicha implementación requiere la inicialización de la biblioteca *Winsock* (i.e. a través de la función *WSAStartup*). Si en el constructor de la clase *Socket* se invocan primitivas de *Winsock*, como ilustra el Listado 2.18, existen dos posibles secuencias de construcción, una de ellas correcta y la otra errónea. Estas secuencias se presentan en la Figura 2.6.

```

1 class Socket
2 {
3     public:

```

ción *_PR_implicitInitialization()*, resultando aproximadamente 100 invocaciones en lugar de 102. Dado que el único objeto del ejemplo es ilustrar la magnitud del impacto, sobre el código fuente, derivado de implementar inicialización implícita, se omite esta corrección.

⁶²Nótese que esta desventaja es propia de todos los métodos de inicialización implícita.

⁶³Aún cuando sea posible predecir o deducir el orden de construcción, resulta difícil garantizar que, en todos los ambientes de compilación, dicho orden se preserve.

```

4     Socket()
5     {
6         _socket = ::socket( ... );
7         if ( _socket == INVALID_SOCKET)
8             throw ...
9     }
10    /* ... */
11
12    protected:
13        SOCKET _socket;
14
15    class Initializer
16    {
17        private:
18            Initializer()
19            {
20                ::WSAStartup( ... );
21            }
22            static Initializer _the_only_instance;
23        public:
24            ~Initializer()
25            {
26                ::WSACleanup();
27            }
28    };
29 };

```

Listado 2.18: Clase de inicialización estática implícita

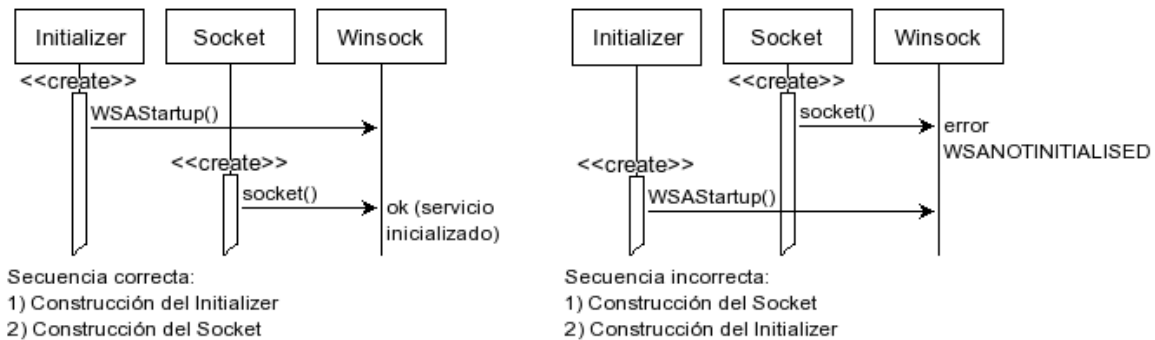


Figura 2.6: Dos secuencias de construcción

En la Figura 2.6 se puede apreciar una secuencia de inicialización incorrecta, donde se construye el *Socket* antes que el *Initializer*. A continuación se ilustra la inconsistencia en la construcción de objetos estáticos con un ejemplo:

```

1 #ifndef _SOCKET_HPP__
2 #define _SOCKET_HPP__
3
4 class Socket
5 {
6     public: Socket();
7     class Initializer
8     {
9         public: ~Initializer();
10        private: Initializer();
11        static Initializer _the_instance;
12    };
13 };

```



```
14
15 #endif
```

Listado 2.19: Interfaz que utiliza el mecanismo de inicialización estática implícita

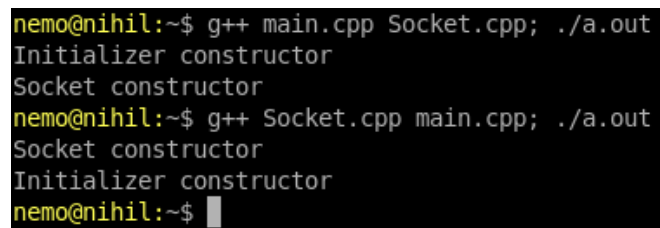
```
1 #include "Socket.hpp"
2
3 #include <stdio.h>
4
5 Socket::Socket() { printf("Socket constructor\n"); }
6
7 Socket::Initializer::~Initializer() {}
8 Socket::Initializer::Initializer() { printf("Initializer constructor\n"); }
9 Socket::Initializer Socket::Initializer::_the_instance;
```

Listado 2.20: Implementación de la interfaz del Listado 2.19

```
1 #include "Socket.hpp"
2 #include <stdio.h>
3
4 Socket sock;
5
6 int main()
7 {
8     return 0;
9 }
```

Listado 2.21: Utilización de la interfaz del Listado 2.19 en una instancia estática

En el Listado 2.19 se especifica una interfaz que utiliza el mecanismo de inicialización estática implícita. La implementación de dicha interfaz se muestra en el Listado 2.20. Tanto el constructor de la clase *Socket*, cuanto el de la clase *Socket::Initializer*, envían una *string* a *stdout*. En la Figura 2.7 se presentan los resultados de compilar el programa en diferente orden. Como puede apreciarse, el orden de construcción de los objetos estáticos depende del orden en el que se especifican los archivos a compilar. En un programa real resulta complicado asegurar el orden de compilación de los archivos de código fuente. Más aún si se utilizan bibliotecas estáticas.



```
nemo@nihil:~$ g++ main.cpp Socket.cpp; ./a.out
Initializer constructor
Socket constructor
nemo@nihil:~$ g++ Socket.cpp main.cpp; ./a.out
Socket constructor
Initializer constructor
nemo@nihil:~$
```

Figura 2.7: Secuencia de construcción dependiente del orden de compilación

Por lo tanto, si el constructor de la clase *Socket* invoca primitivas cuyas precondiciones requieren que el servicio esté inicializado, dichas invocaciones fallarán de forma impredecible, dependiendo de la secuencia de creación de los objetos estáticos. En el caso puntual del ejemplo, es posible evitar este problema modificando el constructor de *Socket* para diferir la invocación a la función *socket()* a otro método. En el Caso de Estudio se ilustra esta solución (ver Listado 3.16).

2.5. Resumen

- La actividad de *Selección de paradigma* consiste en determinar el paradigma en el cual se desarrollará el CPSM. Se relaciona con la *Selección del lenguaje de programación* (ya que el paradigma del CPSM debe estar soportado por el lenguaje de programación del CPSM) y con la *Especificación de las interfaces de cada servicio provisto por el CPSM* (puesto que las interfaces deberán especificarse de acuerdo con el paradigma del CPSM). En la sección 2.1.1 se describieron algunas ventajas y desventajas relacionadas con el paradigma estructurado y el paradigma orientado a objetos.
- La actividad de *Selección del lenguaje de programación* consiste en determinar en qué lenguaje se desarrollará el CPSM. Adicionalmente, el lenguaje de programación del CPSM formará parte del conjunto de servicios trivialmente independientes de la plataforma (i.e. Σ). Esta actividad se relaciona con la *Selección de paradigma*, con la *Definición de plataformas compatibles con el CPSM* (ya que el lenguaje de programación del CPSM deberá estar disponible en todas las plataformas compatibles con el CPSM, i.e. \mathcal{P}) y con la *Definición de servicios provistos por el CPSM* (puesto que la abstracción de los servicios provistos debe poder implementarse en el lenguaje de programación del CPSM). En la sección 2.1.2 se discutió esta actividad en mayor detalle.
- El *Propósito del CPSM* caracteriza y limita el tipo de programas que pueden sustentarse sobre el CPSM. Se distinguen los CPSM de propósito general y los CPSM de propósito específico, siendo sus características predominantes la *Seguridad* y la *Flexibilidad*, respectivamente. En la sección 2.1.3 se describió la influencia que el propósito del CPSM tiene sobre los *Parámetros característicos de un CPSM* (ver sección 1.2). En la subsección 2.1.3.1 se han contrapuesto los parámetros predominantes en los CPSM de propósito general y los CPSM de propósito específico (*Seguridad* y la *Flexibilidad*, respectivamente).
- La actividad de *Definición de servicios provistos por el CPSM* consiste en determinar qué servicios proveerá el CPSM (i.e. \mathcal{S}). Esta actividad es el principal factor determinante de la *Flexibilidad* del CPSM. Se encuentra relacionada con la *Selección del lenguaje de programación* y con la *Definición de plataformas compatibles con el CPSM*. En la sección 2.2 se discutió esta actividad y se presentó la definición de algunos servicios genéricos, utilizados para caracterizar los servicios provistos por diversos CPSM a lo largo de este trabajo (entre ellos el Caso de Estudio, ver sección 3.2). Un aspecto fundamental de la *Definición de servicios provistos por el CPSM* es la *Especificación de las interfaces de cada servicio provisto por el CPSM*. Por *Especificación de las interfaces de cada servicio provisto por el CPSM* se entiende definir con precisión los recursos y primitivas que conforman cada servicio provisto por el CPSM. En la subsección 2.2.1 se presentó los lenguajes de definición de interfaces (IDLs) y el sistema de documentación *Doxygen* como alternativas para especificar las interfaces de los servicios provistos por el CPSM.
- La actividad de *Definición de plataformas compatibles con el CPSM* consiste en determinar con qué plataformas será compatible el CPSM. Esta actividad es el principal factor determinante de la *Compatibilidad* del CPSM. Esta actividad se encuentra relacionada con la *Selección del lenguaje de programación* y con la *Definición de servicios provistos por el CPSM*.

- Una caracterización posible de las implementaciones de un servicio es: “Implementaciones alternativas” e “Implementaciones incompatibles”. Implementaciones alternativas son aquellas que, siendo ambas compatibles con una determinada plataforma, proveen una mecánica distinta de implementación. Implementaciones incompatibles son aquellas para las que existe al menos una plataforma en la que no pueden ejecutarse ambas implementaciones. En la sección 2.4.1 se presentó esta caracterización.
- Por “compilación selectiva” debe entenderse la posibilidad de compilar sólo una porción del código del CPSM, en particular, solo la porción compatible con la plataforma *target*. En la sección 2.4.2 se describió la compilación selectiva. El método de *Compilación condicional* es una manera de resolver el problema de la compilación selectiva utilizando directivas de preprocesador embebidas en el código fuente del CPSM (ver sección 2.4.2.1). Un método alternativo es la *Separación física de archivos y/o directorios*. Este método supone distribuir las implementaciones dependientes de la plataforma en diferentes directorios. Luego determinar qué directorio compilar en función del entorno de compilación (e.g. variables de entorno u opciones al sistema de compilación). En la sección 2.4.2.2 se presentó el método de *Separación física de archivos y/o directorios* y se ilustra el método con un ejemplo sencillo.
- En la sección 2.4.3 se discutió la manera de seleccionar implementaciones alternativas de un servicio. También se presenta una aplicación del patrón *Factory Method*[GHJV95] para seleccionar una instancia que represente la implementación alternativa deseada (en caso de estar disponible en la plataforma subyacente).
- Un problema asociado a la implementación de servicios es el esquema de inicialización y finalización del servicio. El escenario en el cual se presenta este problema consiste en implementaciones incompatibles de un servicio, donde una de dichas implementaciones requiere una inicialización explícita, mientras que la otra se inicializa implícitamente. En la sección 2.4.4 se discutieron ventajas y desventajas de los dos esquemas posibles: inicialización/finalización explícita, e inicialización/finalización implícita. También se presenta un patrón de inicialización implícita que utiliza una instancia estática de una clase de inicialización (i.e. *Initializer*). Dicho patrón provee inicialización y finalización implícitas sin necesidad de realizar comprobaciones en cada primitiva del servicio para determinar si el servicio se encuentra inicializado o se debe invocar un rutina de inicialización implícita.

3. Caso de Estudio

En este capítulo se presenta un caso de estudio completo cuyo objeto es ejemplificar el proceso de desarrollo de un CPSM. La intención del CPSM de estudio no es ser una implementación óptima, en términos de los parámetros presentados en la sección 1.2, sino ilustrar correctamente las decisiones involucradas en el proceso y sus consecuencias.

Esta sección está organizada de la siguiente manera:

- Inicialmente se presenta una descripción del CPSM y del tipo de programa que debe sustentar.
- Luego se define el CPSM en términos del modelo formal (ver sección 1.1) y de las actividades involucradas en su desarrollo (ver Capítulo 2).
- Por último se presenta el desarrollo de una aplicación sustentada sobre el CPSM en varias etapas, a medida que se incorporan los servicios provistos por el CPSM.

3.1. Descripción

El CPSM del Caso de Estudio se propone sustentar el desarrollo de un programa cliente/servidor sobre TCP/IP, donde el comportamiento de la aplicación pueda configurarse en tiempo de ejecución a través de bibliotecas dinámicas externas asociadas explícitamente. El servidor utiliza primitivas sincrónicas (bloqueantes) y atiende concurrentemente a varios clientes. Una posible aplicación del servidor es reenviar los mensajes enviados por un cliente a todos los demás. Esta difusión de mensajes permite realizar una aplicación sencilla de *chat* en modo texto. La lógica de la aplicación (por ejemplo la difusión de mensajes), se programa en una biblioteca independiente del servidor y del cliente, de forma tal que pueda intercambiarse el comportamiento de la aplicación en tiempo de ejecución. El programa, y por lo tanto el CPSM que lo sustenta, debe ser compatible con GNU/Linux y Microsoft Windows. Ambas plataformas se ejecutan sobre una arquitectura de hardware Intel i686 o similar (i.e. AMD). Respecto de las plataformas, se realizarán pruebas en Slackware 10.2, 11 y 12, y en Windows 2000, XP y 2003 Server. Los compiladores a utilizar son GNU g++ (GCC) 3.3.4, 4.1.2 y Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8804 for 80x86.

3.2. Definición del CPSM

El CPSM del Caso de Estudio utiliza un paradigma orientado a objetos, en el lenguaje de programación C++. Como se indica en la Figura 3.1, se considera el estándar de dicho lenguaje[Ame03] como el único servicio trivialmente compatible y punto de articulación entre las plataformas. Los errores serán reportados a través de excepciones y las interfaces se especificarán dentro de los *headers* (*.hpp) utilizando los indicadores presentados en la sección 2.2.1.

El CPSM provee los siguientes servicios (ver sección 2.2 para una descripción más detallada de estos servicios):

- **Concurrencia** (CONC), con los siguientes recursos: *Thread*, *Mutex* y *Condition Variables*.
- **Comunicaciones** (COMM), basado en *BSD Sockets*, únicamente servicio con conexión sobre TCP/IP, cuyo único recurso es *TCP Socket*.

- **Bibliotecas dinámicas (DL)**, cuyo único recurso es *Library*

El CPSM es compatible con dos plataformas: GNU/Linux y Microsoft Windows (las versiones específicas de estas plataformas no se restringen *a priori*).

La abstracción se lleva a cabo a través del mecanismo de separación física de archivos (ver sección 2.4.2.2), delegando en el proceso de compilación la selección de la implementación dependiente de la plataforma. Se utiliza *GNU make* para dirigir el proceso de compilación. En Windows, se utiliza el entorno MSYS como entorno de compilación⁶⁴.

La Figura 3.1 esquematiza el CPSM siguiendo los lineamientos de la Figura 1.1:

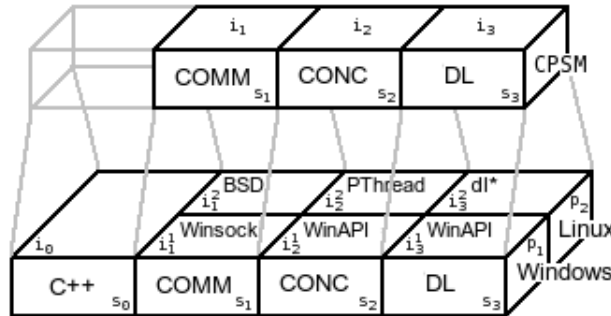


Figura 3.1: Diagrama del Caso de Estudio

La siguiente tabla resume las características del CPSM:

Lenguaje	$s_0 = C++$
Paradigma	Orientado a objetos
Propósito	Caso de Estudio
Plataformas	$\mathcal{P} = \{Linux, Windows\}$
Servicios	$\mathcal{S} = \{s_0, CONC, COMM, DL\}$
Mecanismo de abstracción	Separación física (<i>GNU make</i>)

Nótese que la restricción de punto de articulación entre plataformas (ver sección 1.1) queda satisfecha al verificar que:

$$\Sigma \neq \emptyset$$

O equivalentemente:

$$\exists s_0 = C++ / s_0 \in \Sigma$$

La siguiente tabla resume las tecnologías sobre las cuales se implementan los servicios en cada plataforma:

Servicio	GNU/Linux	Microsoft Windows
Comunicaciones	BSD Sockets	Winsock
Concurrencia	POSIX Threads	WinAPI
Bibliotecas dinámicas	POSIX <i>dl*</i>	WinAPI

⁶⁴Este entorno se utiliza, también, en el ejemplo presentado en la sección 2.4.2.2.

A continuación se presenta la especificación de las interfaces de estos servicios, embebida en los *headers* (*.hpp) del CPSM. Luego se detalla el mecanismo de abstracción de la plataforma subyacente, la estructura de directorios del CPSM, el *Makefile* de compilación, y el entorno de compilación utilizado en cada plataforma.

3.2.1. Especificación de las interfaces de los servicios provistos

En esta sección se especifican las interfaces de los tres servicios provistos. Cada archivo de interfaz incluye un archivo con su mismo nombre precedido por “OS_” (por ejemplo *TCPSocket.hpp* incluye al archivo *OS_TCPSocket.hpp*). Este archivo contiene los detalles dependientes de la plataforma de cada interfaz, entre ellos, la inclusión de los *headers* específicos de la plataforma (e.g., *windows.h*, *netinet/in.h*, etc) y la definición de atributos privados de la interfaz.

3.2.1.1. Comunicaciones

El servicio de Comunicaciones posee un sólo recurso, *TCPSocket*, que encapsula las primitivas de comunicaciones sobre TCP/IP. Se proveen primitivas para escuchar conexiones (*sockets* pasivos) e iniciar conexiones (*sockets* activos). También se proveen primitivas de transmisión y recepción de datos.

```
1 #ifndef _TCPSOCKET_HPP_
2 #define _TCPSOCKET_HPP_
3
4 /**
5  * \file TCPSocket.hpp
6  */
7
8 #include "Exception.hpp"
9
10 // Interfaz dependiente de la plataforma
11 #include "OS_TCPSocket.hpp"
12
13 namespace ce_cpsm {
14
15 /** Representa un @b socket TCP
16  */
17 class TCPSocket
18 {
19 public:
20     /** Construye el @b socket.
21      * \post se contruye el @b socket. El constructor no lanza excepciones.
22      */
23     TCPSocket();
24
25     /** Destruye el @b socket.
26      */
27     ~TCPSocket();
28
29     /** Número máximo de conexiones en cola de espera a ser aceptadas.
30      */
31     static const int BACKLOG_QUEUE_MAX_SIZE;
32
33     /** Tipo de función que se invoca al recibir una conexión mediante
34      * listen().
35      * \param[in] socket el @b socket que representa la nueva conexión.
36      * \return @a false si @p socket no fue utilizado y debe ser destruido,
37      *         @a true en caso contrario (la destrucción de @p socket es
38      *         responsabilidad del usuario, @p socket se debe destruir mediante
```

```
39     *         el operador @b delete).
40 * \pre  @p socket está correctamente creado y conectado, sobre él se
41 *         pueden invocar send(), recv() y close().
42 * \post @p socket no ya no está en uso y puede ser destruido (la función
43 *         devuelve @a false) o @p socket sigue en uso (la función devuelve
44 *         @a true).
45 */
46 typedef bool (*func_t)(TCPSocket* socket);
47
48
49 /** [bloqueante] Espera conexiones en la dirección @p address, extrae la
50 * conexión de la cola de espera del @b socket e invoca la función
51 * @b on_connect.
52 * \param[in] address           dirección donde el @b socket espera
53 *                             conexiones. El valor especial @a NULL será
54 *                             interpretado como ‘‘cualquier dirección’’.
55 * \param[in] port             puerto donde el @b socket espera
56 *                             conexiones. El valor se especifica en
57 *                             formato del host.
58 * \param[in] backlog_queue_size número máximo de conexiones en cola de
59 *                             espera a ser aceptadas. Se espera un valor
60 *                             entre 0 y @a BACKLOG_QUEUE_MAX_SIZE.
61 * \param[in] on_connect       función que se debe llamar al recibir una
62 *                             conexión.
63 * \exception InvalidException el @b socket fue conectado previamente
64 *                             mediante connect().
65 * \exception SecurityException no se tienen permisos suficientes para
66 *                             crear el @b socket.
67 * \exception AddressException la dirección ya se encuentra asignada o no
68 *                             es una dirección válida.
69 * \exception ResourceException no hay recursos suficientes para crear el
70 *                             @b socket (process file table overflow,
71 *                             límite de archivos abiertos alcanzado,
72 *                             memoria insuficiente).
73 * \pre  ningún otro @b socket tiene asignada la dirección y puerto
74 *         representados por @p address y @p port, el @b socket no se ha
75 *         conectado previamente mediante connect().
76 * \post el @b socket escucha conexiones y es capaz de almacenar hasta
77 *         @p backlog_queue_size conexiones en cola de espera. Alcanzado este
78 *         límite las conexiones son ignoradas. Al recibir una conexión se
79 *         llama la función @p on_connect a la cual se le pasa como argumento
80 *         un @b Socket que representa la nueva conexión establecida.
81 */
82 void listen(const char* address,
83            unsigned short port,
84            func_t on_connect,
85            unsigned int backlog_queue_size = BACKLOG_QUEUE_MAX_SIZE)
86     throw (InvalidException,
87           SecurityException,
88           AddressException,
89           ResourceException);
90
91 /** Establece una conexión con el @b socket pasivo que espera conexiones
92 * en la dirección @p address y en el puerto @p port.
93 * \param[in] address dirección del par al cual conectarse.
94 * \param[in] port     puerto del par al cual conectarse.
95 * \exception InvalidException el @b socket fue conectado previamente.
96 * \exception AddressException la dirección es inválida.
97 * \exception ConnectException la conexión fue rechazada, no existe una
98 *                             ruta adecuada al host, se ha excedido el
99 *                             tiempo de espera máximo de establecimiento
100 *                             de la conexión.
101 * \exception ResourceException no hay recursos suficientes para crear el
102 *                             @b socket (process file table overflow,
103 *                             límite de archivos abiertos alcanzado,
104 *                             memoria insuficiente, o no hay puertos
```

```
105     *             locales disponibles).
106     * \post el @b socket se encuentra conectado y permite la transmisión y
107     *     recepción de datos mediante send() y recv(), respectivamente.
108     */
109 void connect(const char* address,
110             unsigned short port)
111     throw (InvalidException,
112           ConnectException,
113           ResourceException);
114
115 /** [bloqueante] Envía un mensaje al @b socket conectado.
116     * \param[in] buffer buffer a enviar.
117     * \param[in] size tamaño en bytes del @p buffer a enviar.
118     * \return cantidad de bytes de @p buffer enviados correctamente.
119     * \exception InvalidException el @b socket no está conectado, o
120     *     @p buffer es @a NULL.
121     * \exception ConnectException se ha cerrado la conexión (interrumpiendo
122     *     la operación).
123     * \exception ResourceException no hay recursos suficientes para enviar el
124     *     mensaje.
125     * \pre el @b socket está conectado mediante connect() o fue pasado por
126     *     parámetro a on_connect(), y no ha sido cerrado.
127     * \post se devuelve el número de bytes enviados correctamente al par
128     *     conectado.
129     */
130 unsigned long send(const void* buffer,
131                  unsigned long size)
132     throw (InvalidException,
133           ConnectException,
134           ResourceException);
135
136 /** [bloqueante] Recibe un mensaje del @b socket conectado.
137     * \param[out] buffer buffer donde se debe copiar el mensaje recibido.
138     * \param[in] size tamaño en bytes del @p buffer de recepción.
139     * \param[in] block si es @a true, la operación bloquea, en caso
140     *     contrario, la operación no bloquea.
141     * \return cantidad de bytes recibidos y copiados en @p buffer.
142     * \exception InvalidException el @b socket no está conectado, o
143     *     @b buffer es NULL.
144     * \exception ConnectException se ha cerrado la conexión (interrumpiendo
145     *     la operación).
146     * \exception ResourceException no hay recursos suficientes para enviar el
147     *     mensaje.
148     * \pre el @b socket está conectado mediante connect() o fue pasado por
149     *     parámetro a on_connect(), y no ha sido cerrado.
150     * \post se copia en @p buffer los bytes leídos y se devuelve el número de
151     *     bytes recibidos correctamente del par conectado.
152     */
153 unsigned long recv(void* buffer,
154                  unsigned long size,
155                  bool block = true)
156     throw (InvalidException,
157           ConnectException,
158           ResourceException);
159
160 /** Desconecta el @b socket y libera los recursos asociados al mismo.
161     * \post se cierra la conexión y se liberan los recursos asociados al
162     *     @b socket, luego de esta llamada no deben invocarse otras
163     *     operaciones sobre el @b socket (o se lanzará una InvalidException).
164     * \exception InvalidException el @b socket no está conectado.
165     */
166 void close() throw (InvalidException);
167
168 // Interfaz dependiente de la plataforma
169 OS_TCPSOCKET_INTERFACE
170 };
```



```
171
172 } // namespace ce_cpsm
173
174 #endif
```

Listado 3.1: cpsm/TCPSocket.hpp: interfaz del servicio de Comunicaciones

3.2.1.2. Concurrency

El servicio de Concurrency posee tres recursos:

- *Thread* encapsula las primitivas que permiten ejecutar código en otro *thread* y aguardar a que un *thread* complete su ejecución. Se proveen tres versiones distintas de *threads*:
 - *ThreadVoid* es un *worker thread* que permite ejecutar una función en otro *thread*. Dicha función recibe como parámetro una variable del tipo *void**.
 - *Thread* es versión *template* de *worker thread* similar a *ThreadVoid*, pero que permite especificar el tipo del parámetro de la función a ejecutar.
 - *ThreadObject* ejecuta el método *run()* en otro *thread*. Para utilizar un *ThreadObject* es necesario heredar una clase de él y redefinir el método *run()*.
- *Mutex* encapsula las primitivas de bloqueo y liberación de *mutexes*.
- *Condition* encapsula las primitivas de *Condition Variables* que permiten aguardar una condición y notificar que una condición se ha cumplido.

Las interfaces de estos recursos se especifican a continuación:

```
1 #ifndef _THREAD_HPP_
2 #define _THREAD_HPP_
3
4 /**
5  * \file Thread.hpp
6  */
7
8 #include "Exception.hpp"
9
10 // Interfaz dependiente de la plataforma
11 #include "OS_Thread.hpp"
12
13 namespace ce_cpsm {
14
15 /** Ejecuta una función que recibe por parámetro un void* en un @b thread
16  * propio.
17  */
18 class ThreadVoid
19 {
20     public:
21         typedef void (*func_t)(void* arg); ///< función principal del @b thread
22
23         ThreadVoid();
24         ~ThreadVoid();
25
26         /** Ejecuta @p f en otro thread con @p arg como argumento.
27          * \param[in] f Función a ejecutar en un nuevo thread.
28          * \param[in] arg Argumento enviado a la función @p f.
29          * \exception ResourceException no hay recursos suficientes para
30          *                               crear el @b thread.
31          * \pre el thread no se ha iniciado previamente.
```

```
32     * \post el thread se ha creado y ejecutará la función @p f con
33     *     el argumento @p arg.
34     */
35     void start(func_t f, void* arg) throw (ResourceException);
36
37     /** Espera a que finalice la ejecución del @b thread iniciado mediante
38     *     start().
39     * \exception InvalidException el @b thread no se ha iniciado o existe
40     *     otro @b thread que ha invocado join().
41     * \pre el @b thread se ha iniciado previamente y no se ha invocado
42     *     join().
43     * \post el @b thread ha finalizado su ejecución.
44     */
45     void join() throw (InvalidException);
46
47     // Interfaz dependiente de la plataforma
48     OS_THREAD_INTERFACE
49 };
50
51 /** Ejecuta una función que recibe por parámetro un ARG_TYPE en un @b thread
52 * propio.
53 */
54 template <typename ARG_TYPE>
55 class Thread
56 {
57     public:
58         typedef void (*func_t)(ARG_TYPE arg); ///< función principal del @b thread
59
60         Thread() {}
61         ~Thread() {}
62
63         /** Ejecuta @p f en otro thread con @p arg como argumento.
64         * \param[in] f Función a ejecutar en un nuevo thread.
65         * \param[in] arg Argumento enviado a la función @p f.
66         * \exception ResourceException no hay recursos suficientes para
67         *     crear el @b thread.
68         * \pre el thread no se ha iniciado previamente.
69         * \post el thread se ha creado y ejecutará la función @p f con
70         *     el argumento @p arg.
71         */
72         void start(func_t f, ARG_TYPE arg) throw (ResourceException)
73         {
74             _f = f;
75             _arg = arg;
76             _imp.start(_inner_function, (void*)this);
77         }
78
79         /** Espera a que finalice la ejecución del @b thread iniciado mediante
80         *     start().
81         * \exception InvalidException el @b thread no se ha iniciado o existe
82         *     otro @b thread que ha invocado join().
83         * \pre el @b thread se ha iniciado previamente y no se ha invocado
84         *     join().
85         * \post el @b thread ha finalizado su ejecución.
86         */
87         void join() throw (InvalidException) { _imp.join(); }
88
89     protected:
90         ThreadVoid _imp;
91         func_t _f;
92         ARG_TYPE _arg;
93         static void _inner_function(void* arg)
94         {
95             Thread<ARG_TYPE>* instance = reinterpret_cast<Thread<ARG_TYPE>*>(arg);
96             if (!instance)
97                 return;
```

```

98
99         (*instance->_f)(instance->_arg);
100     }
101 };
102
103 /** Ejecuta el método run() en un @b thread propio.
104 */
105 class ThreadObject
106 {
107     public:
108         ThreadObject() {}
109         virtual ~ThreadObject() {}
110
111         /** Ejecuta run() en otro thread.
112          * \exception ResourceException no hay recursos suficientes para
113          *                               crear el @b thread.
114          * \pre el thread no se ha iniciado previamente.
115          * \post el thread se ha creado y ejecutará el método run().
116          */
117         void start() throw (ResourceException)
118         {
119             _imp.start(_inner_function, (void*)this);
120         }
121
122         /** Espera a que finalice la ejecución del @b thread iniciado mediante
123          * start().
124          * \exception InvalidException el @b thread no se ha iniciado o existe
125          *                               otro @b thread que ha invocado join().
126          * \pre el @b thread se ha iniciado previamente y no se ha invocado
127          *       join().
128          * \post el @b thread ha finalizado su ejecución.
129          */
130         void join() throw (InvalidException) { _imp.join(); }
131
132         /** Método que será ejecutado en otro @b thread luego de la invocación
133          * a start().
134          * \post el @b thread finalizará su ejecución.
135          */
136         virtual void run() throw () = 0;
137
138     protected:
139         ThreadVoid _imp;
140         static void _inner_function(void* arg)
141         {
142             ThreadObject* instance = reinterpret_cast<ThreadObject*>(arg);
143             if (!instance)
144                 return;
145
146             instance->run();
147         }
148 };
149
150
151 } // namespace ce_cpsm
152
153 #endif

```

Listado 3.2: cpms/Thread.hpp: interfaz del recurso *Thread* del servicio de Concurrency

```

1 #ifndef _MUTEX_HPP__
2 #define _MUTEX_HPP__
3
4 /**
5  * \file Mutex.hpp
6  */
7

```

```

8 #include "Exception.hpp"
9
10 // Interfaz dependiente de la plataforma
11 #include "OS_Mutex.hpp"
12
13
14 namespace ce_cpsm {
15
16 class Condition;
17
18 /** Representa un @b mutex
19  */
20 class Mutex
21 {
22     public:
23
24     /** Constructor.
25      * \exception ResourceException no fue posible crear el @b mutex
26      */
27     Mutex() throw (ResourceException);
28
29     ~Mutex();
30
31     /** [bloqueante] Bloquear el mutex (entrar en la sección crítica).
32      * \exception InvalidException el @b mutex ya se encuentra bloqueado
33      *                                     por este @b thread.
34      * \pre el @b mutex está libre o se encuentra bloqueado por otro
35      *       @b thread.
36      * \post el @b thread que invoco lock() posee el @b mutex.
37      */
38     void lock() throw (InvalidException);
39
40     /** Liberar el mutex (salir de la sección crítica).
41      * \exception InvalidException el @b mutex no se encuentra bloqueado
42      *                                     o no le pertenece a este @b thread.
43      * \pre el @b thread que invoco unlock() posee el @b mutex.
44      * \post el @b mutex está libre o bloqueado por otro @b thread.
45      */
46     void unlock() throw (InvalidException);
47
48     /** Scoped-locking pattern
49      */
50     class Guard
51     {
52     public:
53
54         Guard(Mutex& m) : _m(m) { _m.lock(); }
55         ~Guard()                throw () { try { _m.unlock(); } catch(...) {} }
56     protected:
57         Mutex& _m;
58     };
59
60     friend class Condition;
61
62     // Interfaz dependiente de la plataforma
63     OS_MUTEX_INTERFACE
64 };
65
66 } // namespace ce_cpsm
67
68 #include "cpsm/Condition.hpp"
69
70 #endif

```

Listado 3.3: cpsm/Mutex.hpp: interfaz del recurso *Mutex* del servicio de Concurrencia

```
1 #ifndef _CONDITION_HPP_
2 #define _CONDITION_HPP_
3
4 /**
5  * \file Condition.hpp
6  */
7
8 #include "Exception.hpp"
9 #include "cpsm/Mutex.hpp"
10
11 // Interfaz dependiente de la plataforma
12 #include "OS_Condition.hpp"
13
14 namespace ce_cpsm {
15
16 /** Representa una @b ConditionVariable
17  */
18 class Condition
19 {
20     public:
21
22     /** Constructor.
23      * \param[in] mutex el @b mutex que se debe asociar a la @b condition
24      * \exception ResourceException no fue posible crear la @b condition
25      */
26     Condition(Mutex& mutex) throw (ResourceException);
27
28     ~Condition();
29
30     /** [bloqueante] Suspender el thread hasta que se ejecute signal().
31      * \exception InvalidException el @b mutex asociado a la @b condition
32      * no se encuentra bloqueado por este
33      * @b thread.
34      * \pre el @b mutex con el que se construyó la instancia de Condition
35      * debe estar bloqueado por el @b thread que invocó wait().
36      * \post el @b thread es suspendido hasta que se invoque signal(), al
37      * reanudar su ejecución el mutex seguirá bloqueado por el
38      * @b thread.
39      */
40     void wait() throw (InvalidException);
41
42     /** Reanudar la ejecución de un @b thread suspendido en wait().
43      * \post si uno o mas @b threads están suspendidos en wait(), se
44      * reanudará la ejecución de uno de ellos. Al reanudarse, el
45      * @b thread tendrá control sobre el @b mutex asociado a la
46      * @b condition .
47      */
48     void signal() throw ();
49
50     // Interfaz dependiente de la plataforma
51     OS_CONDITION_INTERFACE
52 };
53
54 } // namespace ce_cpsm
55
56 #endif
```

Listado 3.4: cpsm/Condition.hpp: interfaz del recurso *Condition* del servicio de Concurrency

3.2.1.3. Bibliotecas dinámicas

El servicio de Bibliotecas dinámicas posee un sólo recurso, *Library*, que encapsula la asociación explícita de bibliotecas dinámicas. Dentro de las operaciones que provee se encuentran

la búsqueda de un símbolo por nombre y la búsqueda de una función (de prototipo parametrizable) por nombre.

```
1 #ifndef _LIBRARY_HPP__
2 #define _LIBRARY_HPP__
3
4 /**
5  * \file Library.hpp
6  */
7
8 #include "Exception.hpp"
9
10 // Interfaz dependiente de la plataforma
11 #include "OS_Library.hpp"
12
13 namespace ce_cpsm {
14
15 /** Representa una biblioteca dinamica asociada explicitamente en tiempo de
16  * ejecucion.
17  */
18 class Library
19 {
20     public:
21         /** Cargar la biblioteca en memoria.
22          * \param[in] filename Nombre de la biblioteca que se desea cargar.
23          * \exception ResourceException No fue posible cargar la biblioteca.
24          * \post la biblioteca está cargada en memoria y es posible buscar
25          *         símbolos dentro de ella mediante "findSymbol()".
26          */
27         Library(const char* filename) throw (ResourceException);
28
29         /** Descargar la biblioteca de memoria.
30          * \post la biblioteca se ha descargado y futuras invocaciones al
31          *         método findSymbol() fallarán.
32          */
33         ~Library();
34
35         /** Buscar un símbolo en la biblioteca.
36          * \param[in] name nombre del símbolo (variable, función) que se
37          *                 desea obtener.
38          * \exception ResourceException No fue posible resolver el símbolo
39          *                 @p name.
40          * \return un puntero al símbolo al símbolo encontrado.
41          */
42         void* findSymbol(const char* name) throw (ResourceException);
43
44         // Interfaz dependiente de la plataforma
45         OS_LIBRARY_INTERFACE
46 };
47
48 /** Buscar una función en la biblioteca.
49  * \param[in] FUNCTION_TYPE prototipo de la función que se desea
50  *                 obtener.
51  * \param[in] l biblioteca que contiene la función que se desea obtener.
52  * \param[in] name nombre de la función que se desea obtener.
53  * \exception ResourceException No fue posible resolver el símbolo
54  *                 @p name.
55  * \return un puntero a la función encontrada.
56  */
57 template<typename FUNCTION_TYPE>
58 FUNCTION_TYPE findFunction(Library& l, const char* name) throw
59 (ResourceException)
60 {
61     union
62     {
63         void* as_object;
```

```

64     FUNCTION_TYPE as_function;
65 } inner_cast;
66 inner_cast.as_object = l.findSymbol(name);
67 return inner_cast.as_function;
68 }
69
70 } // namespace ce_cpsm
71
72 #endif

```

Listado 3.5: cpsm/Library.hpp: interfaz del servicio de Bibliotecas dinámica

En la especificación precedente deben remarcarse dos puntos interesantes. En primer lugar, la función *findFunction* no forma parte de la clase *Library* aunque la funcionalidad que provee es similar al método *findSymbol*. El compilador utilizado para compilar el CPSM en Microsoft Windows (Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8804 for 80x86) no soporta correctamente el estándar de C++[Ame03], en particular no soporta métodos parametrizables (i.e. *template*) en clases no parametrizables. Una manera de evitar esta limitación es transformar la definición del método *findFunction* en una función externa a la clase *Library* que reciba una instancia de *Library* como parámetro. Una solución alternativa podría ser no soportar un compilador tan antiguo. En el Caso de Estudio, se optó por soportar la compilación en dicho compilador para ilustrar un caso problemático donde el supuesto del estándar del lenguaje como punto de articulación no es válido. En rigor, el punto de articulación es el subconjunto del estándar soportado por los compiladores utilizados.

El segundo punto de interés es la necesidad de utilizar la construcción *inner_cast* para transformar el resultado del método *findSymbol* en un puntero a función del tipo *FUNCTION_TYPE*. Existe un conflicto entre la interfaz de la función *dlsym*, definida en el estándar POSIX[IEE04], y la conversión entre variables del tipo *void** y punteros a funciones, definida en el estándar de lenguaje C++[Ame03]. Dentro de los tópicos pendientes de definición en el estándar de lenguaje C++, se encuentra dar una solución a ese tipo de conversión[The08a, issue 573][The08b, issue 195]. Una solución posible a este problema es la que se presenta en el Listado 3.5, inspirada en [Pet04]. Esta solución implica realizar la conversión a través de un *union* y funciona bajo el supuesto de que existe una manera válida de realizar la conversión, preservando los patrones de bits involucrados⁶⁵.

3.2.1.4. Servicios comunes

Además de los servicios precedentes, que resultan de la abstracción de los servicios provistos por las plataformas subyacentes, un CPSM puede proveer servicios comunes, cuya implementación no depende la plataforma. El CPSM del Caso de Estudio provee una jerarquía de clases de la familia *Exception* que tipifica los posibles errores asociados a las operaciones de las interfaces del CPSM. La Figura 3.2 ilustra dicha jerarquía.

La declaración de las excepciones del CPSM se encuentra en el archivo “*include/cpsm/Exception.hpp*” que se muestra a continuación:

```

1 #ifndef _EXCEPTION_HPP__
2 #define _EXCEPTION_HPP__
3
4 #include <string>

```

⁶⁵Este no es el caso de la conversión entre variables del tipo *void** y punteros a métodos (i.e. *pointer-to-member-functions*[Ame03]) ya que el tamaño de los punteros a métodos varía en función de las características de la clase apuntada (herencia simple, múltiple, virtual, etc.), y del compilador utilizado[Clu05].

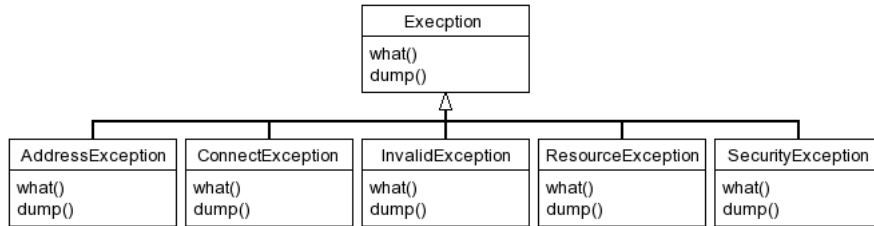


Figura 3.2: Jerarquía de excepciones del CPSM

```

5 #include <stdexcept>
6
7 namespace ce_cpsm {
8
9 class Exception : public std::exception
10 {
11     public:
12         Exception(const char* msg);
13         virtual ~Exception() throw();
14
15         virtual const char* what() const throw ();
16
17         void dump() const;
18
19     protected:
20         std::string _msg;
21 };
22
23 #define DEFINE_EXCEPTION(ex_name) \
24 class ex_name : public Exception \
25 { \
26     public: ex_name(const char* msg) : Exception(#ex_name": ") { _msg += msg;}\
27 }
28
29 DEFINE_EXCEPTION(SecurityException);
30 DEFINE_EXCEPTION(AddressException);
31 DEFINE_EXCEPTION(ResourceException);
32 DEFINE_EXCEPTION(InvalidException);
33 DEFINE_EXCEPTION(ConnectException);
34 DEFINE_EXCEPTION(NotFoundException);
35
36 } // namespace ce_cpsm
37
38 #endif
  
```

Listado 3.6: cpsm/Exception.hpp: excepciones del CPSM

La *macro* `DEFINE_EXCEPTION` define una excepción que hereda de `Exception` y expone un constructor con un único parámetro `const char*` similar al de `Exception`. Los mensajes de las excepciones definidas a través de `DEFINE_EXCEPTION` se prefijan con el nombre de la excepción. Por ejemplo, el mensaje asociado a la excepción arrojada en el Listado 3.7 será “ResourceException: Memoria insuficiente”:

```
throw ResourceException("Memoria insuficiente");
```

Listado 3.7: Ejemplo de uso de excepciones

3.2.2. Mecanismo de abstracción

La abstracción de la plataforma subyacente en el CPSM del Caso de Estudio se lleva a cabo mediante el método de separación física de archivos y directorios (ver sección 2.4.2.2). La estructura de directorios del CPSM se muestra en la Figura 3.3.

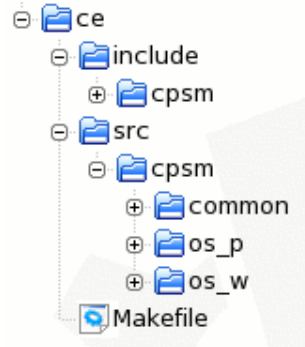


Figura 3.3: Estructura de directorios del CPSM

El CPSM cuenta con los siguientes archivos y directorios:

- *Makefile*: archivo que define el proceso de compilación.
- *include/cpsm/*: directorio que contiene las interfaces de todos los servicios provistos por el CPSM (i.e. todos los archivos correspondientes a los listados presentados en la sección 3.2.1).
- *src/cpsm/common/*: directorio que contiene la implementación de operaciones y recursos independientes de la plataforma.
- *src/cpsm/os_p/*: directorio que contiene la implementación de los servicios provistos por el CPSM, en GNU/Linux.
- *src/cpsm/os_w/*: directorio que contiene la implementación de los servicios provistos por el CPSM, en Microsoft Windows.

El archivo *Makefile* utiliza la misma estrategia que se presentó en el listado 2.8 para determinar la plataforma *target*. Puntualmente, sobre la base del valor de la variable de entorno *OSTYPE* (presente tanto en GNU/Linux cuanto en MSYS), se define una variable de *make* (i.e. *WINDOWS*) que indica la plataforma *target*.

```

ifeq ($(OSTYPE),msys)
    WINDOWS := 1
endif

```

Listado 3.8: Fragmento de *Makefile*: determinación de la plataforma

Dependiendo del valor de dicha variable, se selecciona el compilador a utilizar y el directorio que contiene las implementaciones dependientes de la plataforma:

```

ifdef WINDOWS
    CXX          :=cl.exe
    ...
else
    CXX          :=g++

```

```

    ...
endif

...

ifdef WINDOWS
OS_DIR :=os_w
else
OS_DIR :=os_p
endif

```

Listado 3.9: Fragmento de *Makefile*: configuración del compilador y directorio de compilación

El primer bloque condicional, controlado por la variable *WINDOWS*, permite configurar las características del compilador en función de la plataforma. En el Listado 3.9 se han suprimido los detalles de esta configuración para simplificar la lectura, en el Listado 3.11 se presenta el *Makefile* completo y ahí pueden apreciarse todos los detalles de configuración de los compiladores utilizados: GNU g++ 3.4.6 y Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8804 for 80x86.

La variable *OS_DIR* determina la plataforma subyacente y tiene dos funciones: seleccionar qué archivos se debe compilar, y especificar el directorio de inclusión de interfaces dependientes de la plataforma:

```

SRCS :=$(wildcard src/cpsm/$(OS_DIR)/*.cpp) $(wildcard src/cpsm/common/*.cpp)
OBJS :=$(SRCS:.cpp=.$(OBJ_SUFFIX))

%.${OBJ_SUFFIX} : %.cpp
    @echo "[CXX]    $@"
    @$(CXX) $(CXXFLAGS) -I include -I src/cpsm/$(OS_DIR) -c $<          \
    $(call CXX_OUT,$@)

```

Listado 3.10: Fragmento de *Makefile*: aplicación de la variable *OS_DIR*

En el Listado 3.10 se definen dos variables y una regla de compilación. La primer variable definida es *SRCS* y contiene todos los archivos de código fuente C++ (*.cpp) en los directorios “*src/cpsm/\$(OS_DIR)/*” (implementaciones dependientes de la plataforma) y “*src/cpsm/common/*” (implementaciones independientes de la plataforma). La segunda variable, *OBJS* contiene los mismos elementos que *SRCS*, pero se reemplaza la extensión de los archivos de código fuente (.cpp) por la extensión de archivos de código objeto en la plataforma (*OBJ_SUFFIX*). La regla de compilación indica que todos los archivos de código objeto (*%.\${OBJ_SUFFIX}*) deben compilarse a partir de un archivo de código fuente con el mismo nombre (*%.cpp*). La secuencia de compilación está compuesta por un *echo*, que indica qué archivo se está compilando, y una invocación al compilador seleccionado según el procedimiento mostrado en el Listado 3.9. Es posible apreciar que la variable *OS_DIR* se utiliza en dos oportunidades. Una de ellas es al definir los archivos de código fuente que deben compilarse (y por lo tanto, los archivos de código objeto que resultarán de la compilación). El otro uso de *OS_DIR* es como directorio explícito de inclusión de archivos en el comando de compilación (i.e. *-I src/cpsm/\$(OS_DIR)*).

Al agregar el directorio “*src/cpsm/\$(OS_DIR)/*” dentro de la ruta de búsqueda del compilador, utilizada para resolver la inclusión de archivos, es posible agregar estado (y comportamiento) dependiente de la plataforma, a las interfaces abstractas del CPSM. En todos los listados presentados en la sección 3.2.1 (e.g. Listado 3.1, Listado 3.2, etc) se incluye un archivo cuyo nombre es similar al nombre del archivo de definición de la interfaz con el prefijo “*OS_*”, por ejemplo: *TCPSocket.hpp* incluye a *OS_TCPSocket.hpp*. Los archivos con prefijo “*OS_*” definen la porción dependiente de la plataforma de las interfaces abstractas. Puntualmente, en

ellos se definen las variables que conforman el estado interno de los objetos. Estas variables tienen tipos dependientes de la plataforma. Para completar el ejemplo, ver la sección 3.3.1 donde se presenta el servicio de Comunicaciones del CPSM, tanto en lo que respecta al estado dependiente de la plataforma, cuanto a la implementación del servicio. Adicionalmente, se discute la aplicación del patrón de inicialización estática (ver sección 2.4.4) a la abstracción del servicio de Comunicaciones en Microsoft Windows.

El contenido completo del archivo *Makefile* es:

```

1 ifeq ($(OSTYPE),msys)
2     WINDOWS := 1
3 endif
4
5 ifdef WINDOWS
6     CXX      :=cl.exe
7     CC       :=cl.exe
8     CXX_OUT  :=-Fo'$(1)'
9     CXXFLAGS := -nologo -MD -W3 -GX -O2                \
10             -D 'NDEBUG' -D 'WIN32' -D '_WINDOWS' -D '_MBCS' \
11             -D '_USRDLL' -D 'MSVC' -D '_WIN32_WINNT=0x4000' -FD -c \
12             -nologo
13     LD       :=link.exe
14     LD_OUT   :=-out:$(1)
15     LDFLAGS  := kernel32.lib user32.lib gdi32.lib winspool.lib      \
16             comdlg32.lib advapi32.lib shell32.lib ole32.lib        \
17             oleaut32.lib uuid.lib odbc32.lib odbccp32.lib -nologo \
18             -pdb:none -machine:I386
19     LDFLAGS_SO := -dll
20
21     OBJ_SUFFIX :=.obj
22     SHARED_SUFFIX :=.dll
23     EXE_SUFFIX :=.exe
24 else
25     CXX      :=g++
26     CC       :=gcc
27     CXX_OUT  :=-o $(1)
28     CXXFLAGS :=-pedantic -Wall -Wno-long-long -g
29
30     LD       :=g++
31     LD_OUT   :=-o $(1)
32     LDFLAGS  := $(LDFLAGS) $(CXXFLAGS) -pthread -ldl
33     LDFLAGS_SO := -fno-rtti -fno-exceptions -shared -fPIC
34
35     OBJ_SUFFIX :=.o
36     SHARED_PREFIX :=lib
37     SHARED_SUFFIX :=.so
38 endif
39
40 ifdef WINDOWS
41 OS_DIR :=os_w
42 else
43 OS_DIR :=os_p
44 endif
45
46 SRCS :=$(wildcard src/cpsm/$(OS_DIR)/*.cpp) $(wildcard src/cpsm/common/*.cpp)
47 OBJS :=$(SRCS:.cpp=.$(OBJ_SUFFIX))
48
49 %.$(OBJ_SUFFIX) : %.cpp
50     @echo "[CXX]    $@"
51     @$$(CXX) $(CXXFLAGS) -I include -I src/cpsm/$(OS_DIR) -c $< \
52     $(call CXX_OUT,$@)
53
54 define LINK_EXE
55     @echo "[LD]    $@"
56     @$$(LD) $(LDFLAGS) $^ $(call LD_OUT,$@)

```

```

57 endif
58
59 define LINK_LIBRARY
60     @echo "[LD]      @"
61     @$ (LD) $(LDFLAGS) $(LD_FLAGS_SO) $^ $(call LD_OUT,$@)
62 endif
63
64 define TEST
65     @LD_LIBRARY_PATH=. ./$<
66 endif
67
68 # === BINS =====#
69 # Server1: primer intento, sólo sockets
70 all: Server1$(EXE_SUFFIX)
71 BINS:=$(BINS) Server1$(EXE_SUFFIX)
72 Server1$(EXE_SUFFIX) : src/Server1.$(OBJ_SUFFIX) $(OBJS)
73     $(LINK_EXE)
74
75 # Server2: segundo intento, sockets, threads, mutex y condition
76 all: Server2$(EXE_SUFFIX)
77 BINS:=$(BINS) Server2$(EXE_SUFFIX)
78 Server2$(EXE_SUFFIX) : src/Server2.$(OBJ_SUFFIX) $(OBJS)
79     $(LINK_EXE)
80
81 # Server3: tercer intento, sockets, threads, mutex, condition y library
82 all: Server3$(EXE_SUFFIX)
83 BINS:=$(BINS) Server3$(EXE_SUFFIX)
84 Server3$(EXE_SUFFIX) : src/Server3.$(OBJ_SUFFIX) $(OBJS)
85     $(LINK_EXE)
86
87 # Client
88 all: Client$(EXE_SUFFIX)
89 BINS:=$(BINS) Client$(EXE_SUFFIX)
90 Client$(EXE_SUFFIX) : src/Client.$(OBJ_SUFFIX) $(OBJS)
91     $(LINK_EXE)
92
93
94 all: thread_test$(EXE_SUFFIX)
95 BINS:=$(BINS) thread_test$(EXE_SUFFIX)
96 thread_test$(EXE_SUFFIX) : src/ThreadTest.$(OBJ_SUFFIX) $(OBJS)
97     $(LINK_EXE)
98
99 all: $(SHARED_PREFIX)test.$(SHARED_SUFFIX)
100 BINS:=$(BINS) $(SHARED_PREFIX)test.$(SHARED_SUFFIX)
101 $(SHARED_PREFIX)test.$(SHARED_SUFFIX) : src/DLLTest.$(OBJ_SUFFIX) $(OBJS)
102     $(LINK_LIBRARY)
103
104 all: $(SHARED_PREFIX)protocol1.$(SHARED_SUFFIX)
105 BINS:=$(BINS) $(SHARED_PREFIX)protocol1.$(SHARED_SUFFIX)
106 $(SHARED_PREFIX)protocol1.$(SHARED_SUFFIX) : src/Protocol1.$(OBJ_SUFFIX) $(OBJS)
107     $(LINK_LIBRARY)
108
109 all: $(SHARED_PREFIX)broadcast.$(SHARED_SUFFIX)
110 BINS:=$(BINS) $(SHARED_PREFIX)broadcast.$(SHARED_SUFFIX)
111 $(SHARED_PREFIX)broadcast.$(SHARED_SUFFIX) : src/Broadcast.$(OBJ_SUFFIX) $(OBJS)
112     $(LINK_LIBRARY)
113
114 docs:
115     doxygen doxygen.conf
116
117 .PHONY: clean
118 clean:
119     rm -f src/*.$(OBJ_SUFFIX) $(OBJS) $(BINS) *.idb *.lib *.exp
120     rm -rf docs
121
122 # === TESTS =====#

```

```

123 .PHONY: test
124 test: thread_test$(EXE_SUFFIX) $(SHARED_PREFIX)test.$(SHARED_SUFFIX)
125     $(TEST) $(SHARED_PREFIX)test.$(SHARED_SUFFIX)
126
127 .PHONY: server
128 server: Server3$(EXE_SUFFIX) $(SHARED_PREFIX)broadcast.$(SHARED_SUFFIX)
129     $(TEST) 1234 $(SHARED_PREFIX)broadcast.$(SHARED_SUFFIX)
130
131 .PHONY: client
132 client: Client$(EXE_SUFFIX) $(SHARED_PREFIX)broadcast.$(SHARED_SUFFIX)
133     $(TEST) localhost 1234 $(SHARED_PREFIX)broadcast.$(SHARED_SUFFIX)

```

Listado 3.11: Makefile: archivo director de la compilación del CPSM

3.3. Implementación del CPSM

La implementación del CPSM se circunscribe a los directorios:

- *src/cpsm/common/*: implementación de los servicios comunes, trivialmente independientes de la plataforma ($s_0 \in \Sigma$).
- *src/cpsm/os_p/*: implementación de servicios en GNU/Linux.
- *src/cpsm/os_w/*: implementación de servicios en Microsoft Windows.

La implementación en Windows utiliza el archivo “*src/cpsm/os_w/Utils.hpp*” para facilitar la traducción de errores a excepciones. El contenido de este archivo es:

```

1 #ifndef _UTILS_HPP__
2 #define _UTILS_HPP__
3
4 /**
5  * \file Utils.hpp
6  */
7
8 /** Definir un buffer utilizado por las demás macros para traducir los mensajes
9  * de error a excepciones.
10 */
11 #define WINDOWS_DEFINE_ERROR_BUFFER char _windows_error_buffer[256]
12
13 /** Arrojar la excepción @p ex con el texto correspondiente al último mensaje
14  * de error.
15  * \param[in] ex tipo de excepción arrojar (i.e. InvalidException).
16  */
17 #define WINDOWS_THROW_ERROR(ex) \
18 { \
19     _windows_error_buffer[0] = '\0'; \
20     FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, NULL, GetLastError(), 0, \
21                 _windows_error_buffer, sizeof(_windows_error_buffer)-1, NULL); \
22     throw ex(_windows_error_buffer); \
23 }
24
25 /** Arrojar la excepción @p ex con el texto correspondiente al último mensaje
26  * de error, utilizando un buffer local. (idem WINDOWS_THROW_ERROR pero no
27  * requiere una llamada previa a WINDOWS_DEFINE_ERROR_BUFFER).
28  * \param[in] ex tipo de excepción arrojar (i.e. InvalidException).
29  */
30 #define WINDOWS_THROW(ex) \
31 { \
32     WINDOWS_DEFINE_ERROR_BUFFER; \
33     WINDOWS_THROW_ERROR(ex); \
34 }
35

```

36 `#endif`

Listado 3.12: `cpsm/os_w/Utils.hpp`: *macros* de manipulación de errores en Microsoft Windows

En las siguientes subsecciones, se presenta la implementación de cada servicio provisto por el CPSM. En cada subsección se muestra la definición de estado dependiente de la plataforma (`OS_*.hpp`) y la implementación del servicio, tanto para GNU/Linux cuanto para Microsoft Windows.

3.3.1. Comunicaciones

En GNU/Linux, la implementación de *TCPSocket* se sustenta sobre la API de *sockets* de Linux (incluida en el *kernel*) basada en *BSD Sockets*. El estado dependiente de la plataforma de *TCPSocket* en GNU/Linux es:

```

1 #ifndef _OS_TCPSOCKET_HPP__
2 #define _OS_TCPSOCKET_HPP__
3
4 #include <netinet/in.h>
5
6 #define OS_TCPSOCKET_INTERFACE
7     protected:
8         int _socket;
9         void _createSocket();
10        void _safeClose();
11        void _getAddress(const char* address, unsigned short port, bool
12                        allow_any, struct sockaddr_in& addr);
13
14 #endif

```

Listado 3.13: `cpsm/os_p/OS_TCPSocket.hpp`: estado de la interfaz del servicio de Comunicaciones en GNU/Linux

El atributo protegido `_socket` es el *handle* interno del *socket* nativo de la plataforma (i.e. el *descriptor* devuelto por la función `socket()`). Los métodos declarados en el Listado 3.13 se utilizan para facilitar la implementación, aunque también podrían haberse definido como funciones estáticas en el archivo de implementación del servicio (i.e. `"cpsm/os_p/TCPSocket.cpp"`).

La implementación de *TCPSocket* en GNU/Linux es:

```

1 #include "cpsm/TCPSocket.hpp"
2
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <netdb.h>
6 #include <errno.h>
7 #include <iostream>
8
9 namespace ce_cpsm {
10
11 const int TCPSocket::BACKLOG_QUEUE_MAX_SIZE = SOMAXCONN;
12
13 TCPSocket::TCPSocket() : _socket(-1) {}
14 TCPSocket::~TCPSocket()
15 {
16     _safeClose();
17 }
18
19 void TCPSocket::listen(const char* address, unsigned short port, func_t
20                       on_connect, unsigned int backlog_queue_size) throw (InvalidException,
21                               SecurityException, AddressException, ResourceException)
22 {
23     // 1) crear socket

```

```
24     _createSocket();
25
26     try
27     {
28         // 2) obtener direccion
29         struct sockaddr_in addr;
30         _getAddress(address, port, true, addr);
31
32         // 3) reuse address
33         int val = 1;
34         ::setsockopt(_socket, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
35
36         // 4) asignar direccion
37         if (::bind(_socket, (struct sockaddr*) &addr, sizeof(addr)) == -1)
38         {
39             switch (errno)
40             {
41                 case EACCES:     throw SecurityException("La direccion esta protegida y
42                                 no se tiene acceso a ella");
43                 case EADDRINUSE: throw AddressException("La direccion se encuentra en
44                                 uso");
45                 case ENOMEM:     throw ResourceException("Memoria insuficiente");
46                 default:         throw ResourceException("Error desconocido");
47             }
48         }
49
50         // 5) pasar a modo pasivo
51         if (::listen(_socket, backlog_queue_size) == -1)
52             throw AddressException("La direccion se encuentra en uso");
53
54         // 6) esperar conexiones
55         int new_socket;
56         while ((new_socket = ::accept(_socket, NULL, 0)) != -1)
57         {
58             TCPSocket* s = new TCPSocket;
59             s->_socket = new_socket;
60             if (!(*on_connect)(s))
61                 delete s;
62         }
63         if (errno != EBADF && errno != EINVAL) // errno != socket pasivo cerrado
64         {
65             switch (errno)
66             {
67                 case EMFILE:     throw ResourceException("Se ha alcanzado el limite de
68                                 archivos abiertos por el proceso");
69                 case ENOBUFS:
70                 case ENOMEM:     throw ResourceException("Memoria insuficiente");
71                 default:         throw ResourceException("Error desconocido");
72             }
73         }
74     }
75     catch (Exception& e)
76     {
77         _safeClose();
78         throw;
79     }
80 }
81
82 void TCPSocket::connect(const char* address, unsigned short port) throw
83 (InvalidException, ConnectException, ResourceException)
84 {
85     // 1) crear socket
86     _createSocket();
87
88     try
```

```
87 {
88     // 2) obtener direccion
89     struct sockaddr_in addr;
90     _getAddress(address, port, false, addr);
91
92     // 3) establecer conexion
93     if (::connect(_socket, (struct sockaddr*) &addr, sizeof(addr)) == -1)
94     {
95         switch (errno)
96         {
97             case ECONNREFUSED: throw ConnectException("Conexion rechazada");
98             case EACCES:       throw ConnectException("No se ha podido establecer la
99                             conexion");
100             case EADDRINUSE:  throw AddressException("La direccion se encuentra en
101                             uso");
102             case EAGAIN:      throw ResourceException("No existen puertos locales
103                             disponibles o se excedio el cache de ruteo");
104             case ENETUNREACH: throw ConnectException("No hay ruta al host");
105             case ETIMEDOUT:   throw ConnectException("Tiempo de espera agotado");
106             default:         throw ResourceException("Error desconocido");
107         }
108     }
109 }
110 catch (Exception& e)
111 {
112     _safeClose();
113     throw;
114 }
115 }
116
117 unsigned long TCPSocket::send(const void* buffer, unsigned long size) throw
118 (InvalidException, ConnectException, ResourceException)
119 {
120     ssize_t c;
121     if ((c = ::send(_socket, buffer, size, 0)) == -1)
122     {
123         switch (errno)
124         {
125             case EBADF:
126             case ENOTCONN: throw InvalidException("El socket no esta conectado");
127             case ECONNRESET: throw ConnectException("Se ha cerrado la conexion");
128             case EINVAL:
129             case EFAULT:   throw InvalidException("Buffer invalido");
130             case ENOBUFS:  throw ResourceException("El buffer de la interfaz esta
131                             completo");
132             case ENOMEM:   throw ResourceException("Memoria insuficiente");
133             default:      throw ResourceException("Error desconocido");
134         }
135     }
136 }
137
138 return c;
139 }
140
141 unsigned long TCPSocket::recv(void* buffer, unsigned long size, bool block)
142 throw (InvalidException, ConnectException, ResourceException)
143 {
144     ssize_t c;
145     if ((c = ::recv(_socket, buffer, size, (block) ? 0 : MSG_DONTWAIT)) == -1)
146     {
147         switch (errno)
148         {
149             case EBADF:
150             case ENOTCONN: throw InvalidException("El socket no esta conectado");
151             case ECONNRESET: throw ConnectException("Se ha cerrado la conexion");
152             case EINVAL:
153             case EFAULT:   throw InvalidException("Buffer invalido");
```



```
149         case ENOMEM:         throw ResourceException("Memoria insuficiente");
150         default:             throw ResourceException("Error desconocido");
151     }
152 }
153
154 return c;
155 }
156
157 void TCPSocket::_safeClose()
158 {
159     if (_socket == -1)
160         return;
161
162     ::shutdown(_socket, SHUT_RDWR);
163     ::close(_socket);
164     _socket = -1;
165 }
166
167 void TCPSocket::close() throw (InvalidException)
168 {
169     if (_socket == -1)
170         throw InvalidException("El socket no esta conectado");
171
172     ::shutdown(_socket, SHUT_RDWR);
173
174     if (::close(_socket) == -1)
175         throw InvalidException("Close fallo!");
176
177     _socket = -1;
178 }
179
180 void TCPSocket::_createSocket()
181 {
182     if (_socket != -1)
183         throw InvalidException("El socket esta conectado");
184
185     // 1) crear socket
186     _socket = ::socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
187     if (_socket == -1)
188         switch (errno)
189         {
190             case EACCES: throw SecurityException("No se pudo crear el socket");
191             case EMFILE: throw ResourceException("Se ha alcanzado el limite de archivos
192                 abiertos por el proceso");
193             case ENOBUFFS:
194             case ENOMEM: throw ResourceException("Memoria insuficiente");
195             default:     throw ResourceException("Error desconocido");
196         }
197
198 void TCPSocket::_getAddress(const char* address, unsigned short port,
199 bool allow_any, struct sockaddr_in& addr)
200 {
201     // 2) obtener direccion
202     in_addr_t the_addr;
203     if (address)
204     {
205         struct hostent* ht;
206         while ((ht = ::gethostbyname(address)) == NULL)
207         {
208             if (h_errno == TRY_AGAIN)
209                 continue;
210
211             switch (h_errno)
212             {
```

```

213         case HOST_NOT_FOUND: throw AddressException("No se pudo encontrar una
                direccion IP asociada a address");
214         case NO_ADDRESS: throw AddressException("El valor de address es valido
                pero no tiene una direccion IP asignada");
215         case NO_RECOVERY: throw AddressException("Error en el servidor de
                nombres (DNS)");
216         default: throw AddressException("Error desconocido");
217     }
218 }
219 if (ht->h_length <= 0)
220     throw AddressException("No se pudo resolver address");
221
222 the_addr = *(in_addr_t*)ht->h_addr_list[0];
223 }
224 else
225 {
226     if (allow_any)
227         the_addr = INADDR_ANY;
228     else
229         throw AddressException("Address no puede ser NULL");
230 }
231
232 // 3) construir direccion
233 memset(&addr, 0, sizeof(addr));
234 addr.sin_family = AF_INET;
235 addr.sin_addr.s_addr = the_addr;
236 addr.sin_port = htons(port);
237 }
238
239 } // namespace ce_cpsm

```

Listado 3.14: cpsm/os.p/TCPsocket.cpp: implementación del servicio de Comunicaciones en GNU/Linux

En Microsoft Windows, el servicio de comunicaciones se sustenta sobre la API *Winsock* (versión 2). El estado dependiente de la plataforma de *TCPsocket* en Microsoft Windows es:

```

1 #ifndef _OS_TCPSOCKET_HPP_
2 #define _OS_TCPSOCKET_HPP_
3
4 #include <windows.h>
5
6 #define OS_TCPSOCKET_INTERFACE
7     protected:
8         SOCKET _socket;
9         void _createSocket();
10        void _safeClose();
11        void _getAddress(const char* address, unsigned short port, bool
12            allow_any, struct sockaddr_in& addr);
13        class Initializer
14        {
15        private:
16            Initializer()
17            {
18                WORD wVersionRequested;
19                WSADATA wsaData;
20                wVersionRequested = MAKEWORD( 2, 2 );
21                WSASStartup( wVersionRequested, &wsaData );
22            }
23            static Initializer _the_only_instance;
24        public:
25            ~Initializer()
26            {
27                WSACleanup();
28            }
29        };

```

30
31 #endif

Listado 3.15: cpsm/os.w/OS.TCPSocket.hpp: estado de la interfaz del servicio de Comunicaciones en Microsoft Windows

Nótese que en el Listado 3.15 se utiliza el patrón de inicialización implícita descrito al final de la sección 2.4.4. La principal desventaja que presenta este patrón es la posible inconsistencia en la construcción de objetos estáticos (ver Figura 2.7). Esta restricción se aplica únicamente cuando una precondición del constructor de dichos objetos es que el servicio se encuentre inicializado. En el caso de la implementación de *TCPSocket* en Windows (ver Listado 3.16) el constructor no presenta esta precondición, ya que únicamente inicializa las variables de estado dependiente de la plataforma. La inexistencia de dicha precondición permite evitar los problemas de construcción estática asociados al patrón utilizado.

La implementación de *TCPSocket* en Microsoft Windows es:

```

1 #include "cpsm/TCPSocket.hpp"
2
3 #include <winsock2.h>
4
5 #include <errno.h>
6 #include <iostream>
7
8 #pragma comment(lib, "ws2_32.lib")
9
10 namespace ce_cpsm {
11
12 TCPSocket::Initializer TCPSocket::Initializer::_the_only_instance;
13
14 const int TCPSocket::BACKLOG_QUEUE_MAX_SIZE = SOMAXCONN;
15
16 TCPSocket::TCPSocket() : _socket(INVALID_SOCKET) {}
17 TCPSocket::~TCPSocket()
18 {
19     _safeClose();
20 }
21
22 void TCPSocket::listen(const char* address, unsigned short port, func_t
23     on_connect, unsigned int backlog_queue_size) throw (InvalidException,
24     SecurityException, AddressException, ResourceException)
25 {
26     // 1) crear socket
27     _createSocket();
28
29     try
30     {
31         // 2) obtener direccion
32         struct sockaddr_in addr;
33         _getAddress(address, port, true, addr);
34
35         // 3) reuse address
36         BOOL val = 1;
37         ::setsockopt(_socket, SOL_SOCKET, SO_REUSEADDR, (char*)&val, sizeof(val));
38
39         // 4) asignar direccion
40         if (::bind(_socket, (struct sockaddr*)&addr, sizeof(addr)) == SOCKET_ERROR)
41         {
42             switch (WSAGetLastError())
43             {
44                 case WSAEACCES:     throw SecurityException("La direccion esta
45                                     protegida y no se tiene acceso a ella");
46                 case WSAEADDRINUSE: throw AddressException("La direccion se encuentra
47                                     en uso");

```

```
46         default:                throw ResourceException("Error desconocido");
47     }
48 }
49
50 // 5) pasar a modo pasivo
51 if (::listen(_socket, backlog_queue_size) == SOCKET_ERROR)
52     throw AddressException("La direccion se encuentra en uso");
53
54
55 // 6) esperar conexiones
56 int new_socket;
57 while ((new_socket = ::accept(_socket, NULL, 0)) != INVALID_SOCKET)
58 {
59     TCPSocket* s = new TCPSocket;
60     s->_socket = new_socket;
61     if (!(*on_connect)(s))
62         delete s;
63 }
64 DWORD error = WSAGetLastError();
65 if (error == WSAEINTR) // error = socket pasivo cerrado
66     return;
67 switch (error)
68 {
69     case WSAEMFILE:  throw ResourceException("Se ha alcanzado el limite de
70                     archivos abiertos por el proceso");
71     case WSAENOBUFS: throw ResourceException("Memoria insuficiente");
72     default:        throw ResourceException("Error desconocido");
73 }
74 catch (Exception e)
75 {
76     _safeClose();
77     throw;
78 }
79 }
80
81 void TCPSocket::connect(const char* address, unsigned short port) throw
82 (InvalidException, ConnectException, ResourceException)
83 {
84     // 1) crear socket
85     _createSocket();
86
87     try
88     {
89         // 2) obtener direccion
90         struct sockaddr_in addr;
91         _getAddress(address, port, false, addr);
92
93         // 3) establecer conexion
94         if (::connect(_socket, (struct sockaddr*) &addr, sizeof(addr)) == SOCKET_ERROR)
95         {
96             switch (WSAGetLastError())
97             {
98                 case WSAECONNREFUSED: throw ConnectException("Conexion rechazada");
99                 case WSAEADDRINUSE:   throw AddressException("La direccion se encuentra
100                                     en uso");
101                 case WSAENETUNREACH:  throw ConnectException("No hay ruta al host");
102                 case WSAETIMEDOUT:    throw ConnectException("Tiempo de espera agotado"
103                                     );
104                 default:               throw ResourceException("Error desconocido");
105             }
106         }
107     }
108     catch (Exception e)
109     {
110         _safeClose();
111     }
112 }
```

```
109     throw;
110 }
111 }
112
113 unsigned long TCPSocket::send(const void* buffer, unsigned long size) throw
114     (InvalidException, ConnectException, ResourceException)
115 {
116     int c;
117     if ((c = ::send(_socket, (const char*)buffer, size, 0)) == SOCKET_ERROR)
118     {
119         switch (WSAGetLastError())
120         {
121             case WSAENOTCONN:    throw InvalidException("El socket no esta conectado");
122             case WSAECONNRESET:  throw ConnectException("Se ha cerrado la conexion");
123             case WSAEINVAL:      ;
124             case WSAEFAULT:      throw InvalidException("Buffer invalido");
125             case WSAENOBUFS:     throw ResourceException("El buffer de la interfaz esta
126                                 completo");
127             default:             throw ResourceException("Error desconocido");
128         }
129     }
130     return c;
131 }
132
133 unsigned long TCPSocket::recv(void* buffer, unsigned long size, bool block)
134     throw (InvalidException, ConnectException, ResourceException)
135 {
136     int c;
137     if ((c = ::recv(_socket, (char*)buffer, size, 0)) == SOCKET_ERROR)
138     {
139         switch (WSAGetLastError())
140         {
141             case WSAENOTCONN:    throw InvalidException("El socket no esta conectado");
142             case WSAECONNRESET:  throw ConnectException("Se ha cerrado la conexion");
143             case WSAEINTR:       return 0; // "Socket cerrado localmente"
144             case WSAEINVAL:      ;
145             case WSAEFAULT:      throw InvalidException("Buffer invalido");
146             case WSAENOBUFS:     throw ResourceException("El buffer de la interfaz esta
147                                 completo");
148             default:             throw ResourceException("Error desconocido");
149         }
150     }
151     return c;
152 }
153
154 void TCPSocket::close() throw (InvalidException)
155 {
156     if (_socket == INVALID_SOCKET)
157         throw InvalidException("El socket no esta conectado");
158     ::shutdown(_socket, SD_BOTH);
159     if (::closesocket(_socket) == SOCKET_ERROR)
160         throw InvalidException("Close fallo!");
161     _socket = INVALID_SOCKET;
162 }
163
164 void TCPSocket::_safeClose()
165 {
166     if (_socket == INVALID_SOCKET)
167         return;
168     ::shutdown(_socket, SD_BOTH);
169     ::closesocket(_socket);
170 }
```

```
173     _socket = INVALID_SOCKET;
174 }
175
176 void TCPSocket::_createSocket()
177 {
178     if (_socket != INVALID_SOCKET)
179         throw InvalidException("El socket esta conectado");
180
181     // 1) crear socket
182     _socket = ::socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
183     if (_socket == INVALID_SOCKET)
184         switch (WSAGetLastError())
185         {
186             case WSANOTINITIALISED: throw ResourceException("No se inicializo el
187                                     servicio");
188             case WSAENETDOWN:       throw ResourceException("No se puede acceder a la
189                                     red");
190             case WSAEMFILE:         throw ResourceException("Se ha alcanzado el limite
191                                     de archivos abiertos por el proceso");
192             case WSAENOBUFS:        throw ResourceException("Memoria insuficiente");
193             default:                throw ResourceException("Error desconocido");
194         }
195 }
196
197 void TCPSocket::_getAddress(const char* address, unsigned short port,
198                             bool allow_any, struct sockaddr_in& addr)
199 {
200     // 2) obtener direccion
201     DWORD the_addr;
202     if (address)
203     {
204         struct hostent* ht;
205         while ((ht = ::gethostbyname(address)) == NULL)
206         {
207             if (h_errno == TRY_AGAIN)
208                 continue;
209
210             switch (h_errno)
211             {
212                 case HOST_NOT_FOUND: throw AddressException("No se pudo encontrar una
213                                     direccion IP asociada a address");
214                 case NO_ADDRESS:     throw AddressException("El valor de address es
215                                     valido pero no tiene una direccion IP asignada");
216                 case NO_RECOVERY:    throw AddressException("Error en el servidor de
217                                     nombres (DNS)");
218                 default:             throw AddressException("Error desconocido");
219             }
220         }
221         if (ht->h_length <= 0)
222             throw AddressException("No se pudo resolver address");
223
224         the_addr = *(DWORD*)ht->h_addr_list[0];
225     }
226     else
227     {
228         if (allow_any)
229             the_addr = INADDR_ANY;
230         else
231             throw AddressException("Address no puede ser NULL");
232     }
233
234     // 3) construir direccion
235     memset(&addr, 0, sizeof(addr));
236     addr.sin_family = AF_INET;
237     addr.sin_addr.s_addr = the_addr;
238     addr.sin_port = htons(port);

```

```

233 }
234
235 } // namespace ce_cpsm

```

Listado 3.16: cpsm/os_w/TCPSocket.cpp: implementación del servicio de Comunicaciones en Microsoft Windows

3.3.2. Concurrencia

El servicio de Concurrencia provisto por el CPSM consta de tres recursos: *Thread*, *Mutex* y *Condition*. La implementación de este servicio en GNU/Linux se sustenta en la API de *POSIX Threads*[LB96]. Por lo tanto, los objetos que representan dichos recursos deben contener una variable del tipo *pthread_t*, *pthread_mutex_t* y *pthread_cond_t*, respectivamente.

El estado dependiente de la plataforma de *Thread* en GNU/Linux es:

```

1 #ifndef _OS_THREAD_HPP__
2 #define _OS_THREAD_HPP__
3
4 #include <pthread.h>
5
6 #define OS_THREAD_INTERFACE
7     protected:
8     pthread_t  _thread;
9     func_t    _f;
10    void*      _arg;
11    static void* _thread_start(void* arg);
12
13 #endif

```

Listado 3.17: cpsm/os_p/OS.Thread.hpp: estado de la interfaz de *Thread* (servicio de Concurrencia) en GNU/Linux

Los atributos protegidos *_f* y *_arg* representan la función a ejecutar en otro thread, y el argumento que debe enviarse a esta función, respectivamente. El método estático *_thread_start* es el punto de entrada del nuevo *thread*, creado al ejecutarse *start()* (ver Listado 3.2). El prototipo de dicho método, se ajusta al prototipo de la función *callback* recibida por parámetro en la función *pthread_create*.

La implementación de *Thread* en GNU/Linux es:

```

1 #include "cpsm/Thread.hpp"
2
3 #include <errno.h>
4 #include <iostream>
5
6 namespace ce_cpsm {
7
8 ThreadVoid::ThreadVoid() : _thread(0), _f(NULL) {}
9
10 ThreadVoid::~ThreadVoid()
11 {
12     join();
13 }
14
15 void* ThreadVoid::_thread_start(void* arg)
16 {
17     ThreadVoid* instance = reinterpret_cast<ThreadVoid*>(arg);
18     if (!instance)
19         return NULL;
20
21     (*instance->_f)(instance->_arg);
22

```

```

23     return NULL;
24 }
25
26 void ThreadVoid::start(func_t f, void* arg) throw (ResourceException)
27 {
28     _f = f;
29     _arg = arg;
30     if (pthread_create(&_thread, NULL, _thread_start, this) != 0)
31         throw ResourceException("Recursos insuficientes para crear el thread");
32 }
33
34 void ThreadVoid::join() throw (InvalidException)
35 {
36     if (_thread == 0)
37         return;
38     if (pthread_join(_thread, NULL) == EINVAL)
39         throw InvalidException("Ya existe un thread esperando a este thread");
40     _thread = 0;
41 }
42
43 } // namespace ce_cpsm

```

Listado 3.18: cpsm/os_p/Thread.cpp: implementación de *Thread* (servicio de Concurrencia) en GNU/Linux

Como puede apreciarse en el listado 3.2, la *macro* que define el estado dependiente de la plataforma para *Thread* (i.e. *OS_THREAD_INTERFACE*) se expande dentro de la clase *ThreadVoid*. Como consecuencia, en el Listado 3.18, las primitivas de *Thread* se implementan dentro de la clase *ThreadVoid*. Por ejemplo, la interfaz abstracta de *Thread* declara el método *void join()* (Listado 3.2) y su implementación concreta en GNU/Linux se define como *void ThreadVoid::join()* (Listado 3.18).

El estado dependiente de la plataforma de *Mutex* en GNU/Linux es:

```

1 #ifndef _OS_MUTEX_HPP_
2 #define _OS_MUTEX_HPP_
3
4 #include <pthread.h>
5
6 #define OS_MUTEX_INTERFACE \
7     protected: \
8         pthread_mutex_t _mutex;
9
10 #endif

```

Listado 3.19: cpsm/os_p/OS_Mutex.hpp: estado de la interfaz de *Mutex* (servicio de Concurrencia) en GNU/Linux

El atributo protegido *_mutex* representa el *handle* al *POSIX mutex* interno que provee la funcionalidad de exclusión mutua.

La implementación de *Mutex* en GNU/Linux es:

```

1 #include "cpsm/Mutex.hpp"
2
3 #include <pthread.h>
4
5 namespace ce_cpsm {
6
7 Mutex::Mutex() throw (ResourceException)
8 {
9     pthread_mutex_init(&_mutex, NULL);
10 }
11
12 Mutex::~Mutex()

```



```

13 {
14     pthread_mutex_destroy(&_mutex);
15 }
16
17 void Mutex::lock() throw (InvalidException)
18 {
19     if (pthread_mutex_lock(&_mutex) != 0)
20         throw InvalidException("El Mutex ya esta bloqueado por este thread");
21 }
22
23 void Mutex::unlock() throw (InvalidException)
24 {
25     if (pthread_mutex_unlock(&_mutex) != 0)
26         throw InvalidException("El Mutex no esta bloqueado por este thread");
27 }
28
29 } // namespace ce_cpsm

```

Listado 3.20: cpsm/os_p/Mutex.cpp: implementación de *Mutex* (servicio de Concurrency) en GNU/Linux

El estado dependiente de la plataforma de *Condition* en GNU/Linux es:

```

1 #ifndef _OS_CONDITION_HPP__
2 #define _OS_CONDITION_HPP__
3
4 #include <pthread.h>
5
6 #define OS_CONDITION_INTERFACE \
7     protected: \
8     pthread_cond_t _cond; \
9     Mutex& _mutex;
10
11 #endif

```

Listado 3.21: cpsm/os_p/OS_Condition.hpp: estado de la interfaz de *Condition* (servicio de Concurrency) en GNU/Linux

Los atributos protegidos *_cond* y *_mutex* representan, respectivamente, el *handle* a la *POSIX condition variable* interna que provee la funcionalidad de *Condition Variable*, y el *Mutex* sobre el cual la *Condition Variable* debe actuar. El prototipo de la función *pthread_cond_wait* (que se utiliza para implementar el método *wait()* de *Condition*, ver Listado 3.22) recibe un *handle* a un *POSIX mutex*, este valor se obtiene del atributo protegido *_mutex* de la clase *Mutex*. Para permitir acceso a dicho atributo, fue necesario declarar a la clase *Condition* como *friend* de *Mutex* (ver Listado 3.3).

La implementación de *Condition* en GNU/Linux es:

```

1 #include "cpsm/Condition.hpp"
2
3 #include <pthread.h>
4 #include <errno.h>
5
6 namespace ce_cpsm {
7
8 Condition::Condition(Mutex& mutex) throw (ResourceException) : _mutex(mutex)
9 {
10     switch(pthread_cond_init(&_cond, NULL))
11     {
12         case 0: break;
13         case EAGAIN: throw ResourceException("Recursos insuficientes para crear la
14             Condition");
15         case ENOMEM: throw ResourceException("Memoria insuficiente");
16         default: throw ResourceException("Error desconocido");
17     }
18 }

```

```
17 }
18
19 Condition::~Condition()
20 {
21     pthread_cond_destroy(&_cond);
22 }
23
24 void Condition::wait() throw (InvalidException)
25 {
26     if (pthread_cond_wait(&_cond, &_mutex._mutex) != 0)
27         throw InvalidException("El Mutex asociado no esta bloqueado por este thread");
28 }
29
30 void Condition::signal() throw ()
31 {
32     pthread_cond_signal(&_cond);
33 }
34
35 } // namespace ce_cpsm
```

Listado 3.22: cpsm/os_p/Condition.cpp: implementación de *Condition* (servicio de Concurrencia) en GNU/Linux

En Microsoft Windows, el servicio de Concurrencia se implementa directamente sobre API de Windows (*WinAPI*). La implementación de Thread es relativamente directa y utiliza un *HANDLE* para representar el *thread*. Respecto a los recursos *Mutex* y *Condition* existen algunas alternativas.

Windows provee diversos mecanismos de sincronización sobre los cuales puede construirse un *Mutex* que implemente la interfaz especificada en el Listado 3.3. Entre ellos puede mencionarse *CRITICAL_SECTION* y *Windows Mutex Object* (representado a través de un *HANDLE*).

Cada mecanismo tiene ventajas y desventajas:

- Las *CRITICAL_SECTIONs* resultan mucho más eficientes que los *Windows Mutex Objects*, pudiendo exhibir un aumento relativo de la *Performance* de hasta un orden de magnitud respecto a los *Windows Mutex Objects*[Bin08].
- Los *Windows Mutex Objects* permiten sincronizar threads en diferentes procesos. Puesto que la interfaz abstracta del CPSM para *Mutex* no requiere proveer sincronización entre procesos, esta ventaja no es significativa. Sin embargo, Windows no provee *Condition Variables* por lo que es necesario realizar una implementación dentro del CPSM. En [SP] se analizan varias implementaciones posibles de *Condition Variables* en Windows. La solución recomendada utiliza la primitiva *SignalObjectAndWait*[Mic08b] que tiene dos limitaciones: No es compatible con algunas versiones de Windows 32-bit (Windows 95, 98, NT Version menor a 4), y no puede operar sobre *CRITICAL_SECTIONs*.

Para implementar *Condition Variables* en el CPSM del Caso de Estudio, se utilizó una adaptación de la solución planteada en [SP], que utiliza *SignalObjectAndWait* y *Windows Mutex Objects*. Por lo tanto, el recurso *Mutex* fue implementado utilizando *Windows Mutex Objects*.

Una posible extensión al CPSM consiste en proveer una interfaz de inicialización explícita para el servicio de Concurrencia (ver sección 2.4.4) que permita seleccionar entre implementaciones alternativas de *Mutex* y *Condition*. Una de dichas implementaciones es la provista actualmente en el CPSM y otra podría ser una implementación con *Mutex* basados en *CRITICAL_SECTIONs* y *Condition* implementadas según la solución que utiliza la función *SetEvent*

en [SP]. Esta solución no asegura equidad (*fairness*) en la notificación de *threads*, pero no requiere la utilización de *Windows Mutex Objects*.

El estado dependiente de la plataforma de *Thread* en Microsoft Windows es:

```

1 #ifndef _OS_THREAD_HPP_
2 #define _OS_THREAD_HPP_
3
4 #include <windows.h>
5
6 #define OS_THREAD_INTERFACE
7     protected:
8         HANDLE      _thread;
9         func_t      _f;
10        void*       _arg;
11        static unsigned __stdcall _thread_start(void* arg);
12
13 #endif

```

Listado 3.23: cpsm/os.w/OS.Thread.hpp: estado de la interfaz de *Thread* (servicio de Concurrencia) en Microsoft Windows

Los atributos protegidos de estado dependiente de la plataforma en Microsoft Windows son similares a los presentados en el Listado 3.24 para GNU/Linux. En este caso, el atributo *_thread* es del tipo *HANDLE* en lugar de *int* y el prototipo del método estático *_thread_start* se ajusta al prototipo de la función *callback* recibida por parámetro en la función *_beginthreadex*.

La implementación de *Thread* en Microsoft Windows es:

```

1 #include "cpsm/Thread.hpp"
2 #include "Utils.hpp"
3
4 #include <process.h>
5 #include <stdlib.h>
6
7 namespace ce_cpsm {
8
9 ThreadVoid::ThreadVoid() : _thread(0) {}
10
11 ThreadVoid::~ThreadVoid()
12 {
13     join();
14 }
15
16 unsigned __stdcall ThreadVoid::_thread_start(void* arg)
17 {
18     ThreadVoid* instance = reinterpret_cast<ThreadVoid*>(arg);
19     if (!instance)
20         return 0;
21
22     (*instance->_f)(instance->_arg);
23
24     return 0;
25 }
26
27 void ThreadVoid::start(func_t f, void* arg) throw (ResourceException)
28 {
29     _f = f;
30     _arg = arg;
31
32     if ((_thread = (HANDLE)_beginthreadex(NULL,0,_thread_start,this,0,NULL)) == 0)
33     {
34         if (errno == EAGAIN)
35             throw ResourceException("Recursos insuficientes para crear el thread");
36         else
37             throw ResourceException("Error desconocido");

```

```

38     }
39 }
40
41 void ThreadVoid::join() throw (InvalidException)
42 {
43     if (_thread == 0)
44         return;
45
46     if (WaitForSingleObject(_thread, INFINITE) == WAIT_FAILED)
47         WINDOWS_THROW(InvalidException);
48
49     CloseHandle(_thread);
50     _thread = 0;
51 }
52
53 } // namespace ce_cpsm

```

Listado 3.24: cpsm/os_w/Thread.cpp: implementación de *Thread* (servicio de Concurrencia) en Microsoft Windows

El estado dependiente de la plataforma de *Mutex* en Microsoft Windows es:

```

1 #ifndef _OS_MUTEX_HPP_
2 #define _OS_MUTEX_HPP_
3
4 #include <windows.h>
5
6 #define OS_MUTEX_INTERFACE
7     protected:
8         HANDLE _mutex;
9
10 #endif

```

Listado 3.25: cpsm/os_w/OS_Mutex.hpp: estado de la interfaz de *Mutex* (servicio de Concurrencia) en Microsoft Windows

El atributo protegido *_mutex* representa el *handle* al *Windows Mutex Object* interno que provee la funcionalidad de exclusión mutua.

La implementación de *Mutex* en Microsoft Windows es:

```

1 #include "cpsm/Mutex.hpp"
2 #include "Utils.hpp"
3
4 namespace ce_cpsm {
5
6 Mutex::Mutex() throw (ResourceException)
7 {
8     if ((_mutex = CreateMutex(NULL, FALSE, NULL)) == NULL)
9         WINDOWS_THROW(ResourceException);
10 }
11
12 Mutex::~Mutex()
13 {
14     if (_mutex != NULL)
15         CloseHandle(_mutex);
16 }
17
18 void Mutex::lock() throw (InvalidException)
19 {
20     if (WaitForSingleObject(_mutex, INFINITE) == WAIT_FAILED)
21         WINDOWS_THROW(InvalidException);
22 }
23
24 void Mutex::unlock() throw (InvalidException)
25 {
26     if (ReleaseMutex(_mutex) != TRUE)

```

```

27     WINDOWS_THROW(InvalidException);
28 }
29
30 } // namespace ce_cpsm

```

Listado 3.26: cpsm/os_w/Mutex.cpp: implementación de *Mutex* (servicio de Concurrencia) en Microsoft Windows

El estado dependiente de la plataforma de *Condition* en Microsoft Windows es:

```

1 #ifndef _OS_CONDITION_HPP__
2 #define _OS_CONDITION_HPP__
3
4 #include <windows.h>
5
6 #define OS_CONDITION_INTERFACE \
7     protected: \
8     CRITICAL_SECTION _waiting_cs; \
9     unsigned long _waiting; \
10    HANDLE _sem; \
11    Mutex& _mutex;
12
13 #endif

```

Listado 3.27: cpsm/os_w/OS-Condition.hpp: estado de la interfaz de *Condition* (servicio de Concurrencia) en Microsoft Windows

La implementación de *Condition* en Microsoft Windows es:

```

1 #include "cpsm/Condition.hpp"
2 #include "Utils.hpp"
3
4 #if (_WIN32_WINNT < 0x0400)
5     #ifndef _WIN32_WINNT
6         #error _WIN32_WINNT no esta definida
7     #else
8         #error Version de windows no soportada!
9     #endif
10 #endif
11
12 // Esta implementación es una versión adaptada de la propuesta en
13 // "Strategies for Implementing POSIX Condition Variables on Win32"
14 // de Douglas C. Schmidt y Irfan Pyarali.
15 //
16 // url: http://www.cs.wustl.edu/~schmidt/win32-cv-1.html
17 //
18 // Nota: se preservan los comentarios en inglés de la versión
19 // original.
20
21 namespace ce_cpsm {
22
23 Condition::Condition(Mutex& mutex) throw (ResourceException) : _waiting(0), _mutex(
24     mutex)
25 {
26     InitializeCriticalSection(&_waiting_cs);
27     _sem = CreateSemaphore (NULL, // no security
28                             0, // initially 0
29                             0x7fffffff, // max count
30                             NULL); // unnamed
31 }
32
33 Condition::~Condition()
34 {
35     CloseHandle(_sem);
36     DeleteCriticalSection(&_waiting_cs);
37 }

```

```

38 void Condition::wait() throw (InvalidException)
39 {
40     // Avoid race conditions.
41     EnterCriticalSection (&_waiting_cs);
42     ++_waiting;
43     LeaveCriticalSection (&_waiting_cs);
44
45     // This call atomically releases the mutex and waits on the
46     // semaphore until <pthread_cond_signal> or <pthread_cond_broadcast>
47     // are called by another thread.
48     if (SignalObjectAndWait(_mutex._mutex, _sem, INFINITE, FALSE) == WAIT_FAILED)
49         WINDOWS_THROW(InvalidException);
50
51     // Reacquire lock to avoid race conditions.
52     EnterCriticalSection (&_waiting_cs);
53
54     // We're no longer waiting...
55     _waiting--;
56
57     LeaveCriticalSection (&_waiting_cs);
58
59     // Always regain the external mutex since that's the guarantee we
60     // give to our callers.
61     _mutex.lock();
62
63     //WINDOWS_THROW(InvalidException);
64 }
65
66 void Condition::signal() throw ()
67 {
68     EnterCriticalSection (&_waiting_cs);
69     int have_waiting = _waiting > 0;
70     LeaveCriticalSection (&_waiting_cs);
71
72     // If there aren't any waiters, then this is a no-op.
73     if (have_waiting)
74         ReleaseSemaphore (_sem, 1, 0);
75 }
76
77 } // namespace ce_cpasm

```

Listado 3.28: cpsm/os_w/Condition.cpp: implementación de *Condition* (servicio de Concurrencia) en Microsoft Windows

La implementación de *Condition Variables* presentada en el Listado 3.28 es una versión adaptada de una de las soluciones propuestas en [SP]. Puntualmente se adaptó la solución que utiliza *SignalObjectAndWait* y *Windows Mutex Objects* eliminando la complejidad asociada a la primitiva *broadcast*. Dado que la interfaz de *Condition* (ver Listado 3.4) no especifica dicha primitiva, la adaptación permite simplificar la implementación⁶⁶

3.3.3. Bibliotecas dinámicas

En GNU/Linux, el único recurso del servicio de Bibliotecas dinámicas (*Library*), se implementa utilizando el servicio de asociación dinámica de bibliotecas definido en sistemas POSIX (i.e. la familia de funciones *dl**)[IEE04].

⁶⁶Respecto a la primitiva *broadcast*, su implementación debe considerarse cuidadosamente para evitar efectos secundarios como el *cache line ping-pong*. Este efecto se debe a que todos los *threads* notificados deben bloquear el *Mutex* asociado a la *Condition Variable* y el acceso a la memoria de dichos recursos puede distribuirse a lo largo de las líneas de *cache* de todos los procesadores del sistema. Este efecto genera un intercambio de copias y notificaciones muy costoso, cuando, en última instancia, un único *thread* podrá bloquear el *Mutex*[Dre08].

El estado dependiente de la plataforma de *Library* en GNU/Linux es:

```

1 #ifndef _OS_LIBRARY_HPP__
2 #define _OS_LIBRARY_HPP__
3
4 #define OS_LIBRARY_INTERFACE          \
5     protected:                       \
6         void* _library;
7
8 #endif

```

Listado 3.29: cpsm/os_p/OS.Library.hpp: estado de la interfaz del servicio de Bibliotecas dinámicas en GNU/Linux

El único atributo protegido que conforma el estado dependiente de la plataforma del servicio de Bibliotecas dinámicas en GNU/Linux es *_library*, que representa la biblioteca dinámica asociada mediante *dlopen*.

La implementación de *Library* en GNU/Linux es:

```

1 #include "cpsm/Library.hpp"
2
3 #include <stdlib.h>
4 #include <dlfcn.h>
5
6 namespace ce_cpsm {
7
8 Library::Library(const char* filename) throw (ResourceException)
9 {
10     if ((_library = dlopen(filename, RTLD_LAZY)) == NULL)
11         throw ResourceException(dlerror());
12 }
13
14 Library::~~Library()
15 {
16     dlclose(_library);
17 }
18
19 void* Library::findSymbol(const char* name) throw (ResourceException)
20 {
21     dlerror(); // eliminar errores previos.
22     void* sym = dlsym(_library, name);
23     const char* error = dlerror();
24
25     if (error)
26         throw ResourceException(error);
27
28     return sym;
29 }
30
31 } // namespace ce_cpsm

```

Listado 3.30: cpsm/os_p/Library.cpp: implementación del servicio de Bibliotecas dinámicas en GNU/Linux

En Microsoft Windows, el servicio de Bibliotecas dinámicas se implementa mediante las funciones *LoadLibrary*, *GetProcAddress* y *FreeLibrary*.

El estado dependiente de la plataforma de *Library* en Microsoft Windows es:

```

1 #ifndef _OS_LIBRARY_HPP__
2 #define _OS_LIBRARY_HPP__
3
4 #include <windows.h>
5
6 #define OS_LIBRARY_INTERFACE          \
7     protected:                       \

```

```

8         HMODULE _library;
9
10 #endif

```

Listado 3.31: `cpsm/os_w/OS.Library.hpp`: estado de la interfaz del servicio de Bibliotecas dinámicas en Microsoft Windows

Al igual que en el caso de GNU/Linux, en Microsoft Windows, el estado dependiente de la plataforma para el servicio de Bibliotecas dinámicas se representa mediante el atributo `_library`. La diferencia radica en que, en este caso, el tipo de la variable es `HMODULE` en lugar de `void*`.

La implementación de `Library` en Microsoft Windows:

```

1 #include "cpsm/Library.hpp"
2 #include "Utils.hpp"
3
4 namespace ce_cpsm {
5
6 Library::Library(const char* filename) throw (ResourceException)
7 {
8     if ((_library = LoadLibrary(filename)) == NULL)
9         WINDOWS_THROW(ResourceException);
10 }
11
12 Library::~Library()
13 {
14     FreeLibrary(_library);
15 }
16
17 void* Library::findSymbol(const char* name) throw (ResourceException)
18 {
19     SetLastError(ERROR_SUCCESS); // eliminar errores previos.
20     void* sym = GetProcAddress(_library, name);
21     DWORD error = GetLastError();
22
23     if (error != ERROR_SUCCESS)
24         WINDOWS_THROW(ResourceException);
25
26     return sym;
27 }
28
29 } // namespace ce_cpsm

```

Listado 3.32: `cpsm/os_w/Library.cpp`: implementación del servicio de Bibliotecas dinámicas en Microsoft Windows

3.3.4. Servicios comunes

La implementación de servicios comunes a todas las plataformas no depende de características particulares de ninguna de las plataformas compatibles con el CPSM. Por lo tanto, no es necesario utilizar un mecanismo para declarar estado dependiente de la plataforma como se utilizó en las secciones anteriores.

A continuación se muestra la implementación de la clase `Exception`:

```

1 #include "cpsm/Exception.hpp"
2
3 #include <iostream>
4
5 namespace ce_cpsm {
6
7 Exception::Exception(const char* msg) : _msg(msg) {}
8 Exception::~Exception() throw() {}
9

```



```
10 const char* Exception::what() const throw ()
11 {
12     return _msg.c_str();
13 }
14
15 void Exception::dump() const
16 {
17     std::cerr << _msg << std::endl;
18 }
19
20 } // namespace ce_cpsm
```

Listado 3.33: cpsm/common/Exception.cpp: implementación *Exception* correspondiente a la interfaz del Listado 3.6

Nótese que la *macro* utilizada para definir las excepciones que extienden la clase *Exception* (i.e. *DEFINE_EXCEPTION*, ver Listado 3.6) implementa todos los métodos de la excepción definida. Por lo tanto, en el Listado 3.33 sólo se implementan los métodos de la clase base de la jerarquía, *Exception*.

3.4. Desarrollo de un programa sustentado sobre el CPSM

En la sección 3.1 se describió el tipo de programas sustentables sobre el CPSM del Caso de Estudio. A continuación se presenta un programa que ilustra la utilización de dicho CPSM para lograr la abstracción de la plataforma subyacente. El programa se presenta en tres intentos, cada uno de ellos incorpora nuevos servicios del CPSM para lograr, en el tercer intento, la funcionalidad esperada.

En el primer intento se realiza un servidor sencillo que espera conexiones. Al recibir una conexión, la acepta, envía un mensaje “Hello World” y luego finaliza la conexión. El servidor es iterativo y se ejecuta en un único *thread* (el principal de la aplicación). A modo de cliente se utiliza el programa *telnet*.

El segundo intento incorpora múltiples *threads*. El servidor es concurrente, es decir, cada conexión es atendida en un *thread* separado. El servicio provisto es similar al primer intento, se envía el mensaje “Hello World” y luego se finaliza la conexión. Nuevamente, a modo de cliente se utiliza el programa *telnet*.

En el tercer y último intento se incorpora la asociación explícita de Bibliotecas dinámicas. El servidor es similar al del intento anterior, pero la lógica del servicio provisto se delega en la función *server_protocol_connect()*. El programa cliente establece la conexión con el servidor y ejecuta la función *client_protocol()*. Ambas funciones se importan en tiempo de ejecución de una biblioteca externa, que determina el tipo de interacción entre el servidor y el cliente.

3.4.1. Primer intento: Comunicaciones

El primer intento consiste en un programa servidor que espera conexiones y las atiende iterativamente. Luego de establecida una conexión, el programa envía un mensaje “Hello World” y finaliza la conexión. Dado que el servidor es iterativo, el único servicio del CPSM utilizado es el servicio de Comunicaciones.

En la Figura 3.4 se muestra la secuencia de interacción entre el programa servidor y un cliente. En la misma se utiliza la siguiente convención: cuando un objeto se encuentra en control del *thread* de ejecución, la línea de vida del objeto se representa sólida, de color negro. Por ejemplo, entre la invocación del método *listen()* y el retorno de dicho método (indicado como *on_connect()*), el *thread* principal del servidor se encuentra en control del objeto *PSocket*.

Para representar este caso, la línea de vida de dicho objeto se dibuja sólida en negro entre la invocación de *listen()* y su retorno.

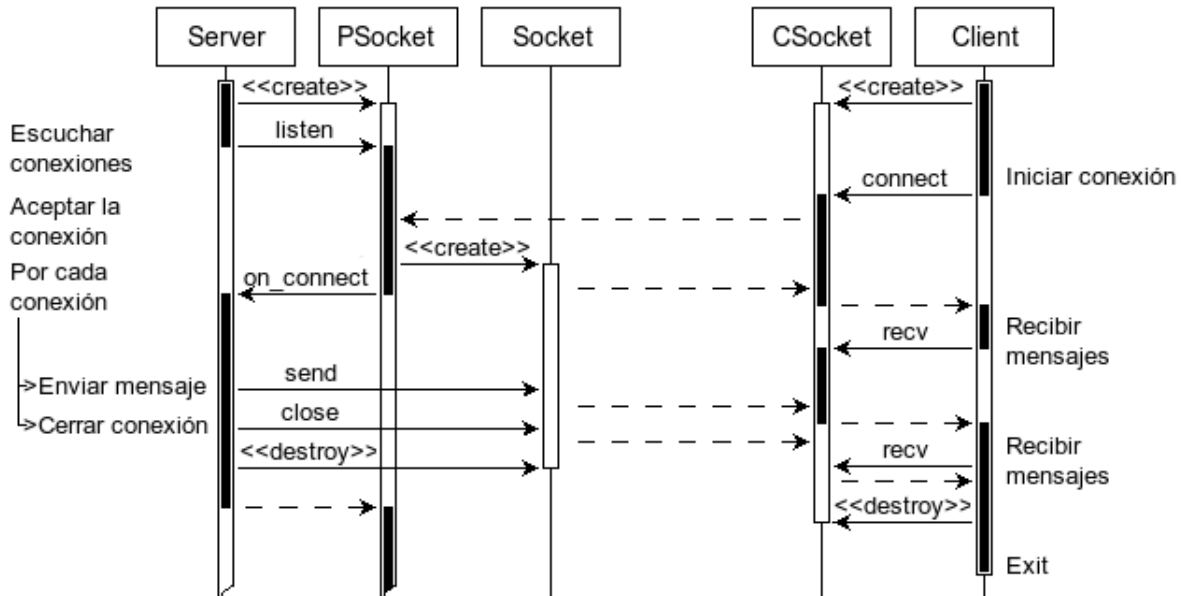


Figura 3.4: Secuencia de interacción para el primer intento

En la Figura 3.4, los objetos interactuantes son, de izquierda a derecha:

- **Server:** la lógica principal de ejecución del programa servidor, representada por el comportamiento estático de la clase *Server*.
- **PSocket:** el *socket* pasivo que escucha conexiones en el programa servidor.
- **Socket:** el *socket* resultante de aceptar un conexión en *PSocket*.
- **CSocket:** el *socket* activo, a través del cual, el programa cliente establece la conexión.
- **Cliente:** la lógica principal de ejecución del programa cliente.

La secuencia del programa servidor puede resumirse de esta manera: El servidor (*Server*) construye un *socket* pasivo (*PSocket*) y espera las conexiones que arriben en dicho *socket* (*listen()*). Por cada conexión establecida (*on_connect()*), se envía el mensaje “Hello World” (*send()*) a través del *socket* que representa la conexión (*Socket*) y luego se finaliza la conexión (*close()*).

La secuencia del programa cliente es una versión simplificada del comportamiento obtenido al establecer una conexión mediante el programa *telnet*.

A continuación se muestra el código del programa servidor en este intento, denominado *Server1.cpp*:

```

1 #include "cpsm/TCPsocket.hpp"
2 #include <iostream>
3
4 namespace cpsm = ce_cpsm;
5

```

```
6 /** La clase Server agrupa el comportamiento y estado del servidor.
7  * El método principal de Server, startup(), inicia el servidor y
8  * escucha conexiones en el @a server_socket .
9  *
10 * El método on_connect() es invocado por cada conexión entrante.
11 */
12 class Server
13 {
14     public:
15         /// Función callback invocada por cada conexión entrante
16         static bool on_connect(cpsm::TCPSocket* s) throw ()
17         {
18             try
19             {
20                 std::cout << "on_connect" << std::endl;
21
22                 // Enviar mensaje
23                 const char msg[] = "\n\n*** Hello World! ***\n\n\n";
24                 s->send(msg, sizeof(msg));
25                 delete s;
26
27                 std::cout << "on_connect done" << std::endl;
28
29                 return true; // el socket fue utilizado
30             }
31             catch (cpsm::Exception& e)
32             {
33                 e.dump();
34                 return false; // destruir socket
35             }
36         }
37
38         /// Función principal del servidor
39         static void startup(unsigned short port) throw ()
40         {
41             // Crear socket pasivo y esperar conexiones
42             cpsm::TCPSocket server_socket;
43             try
44             {
45                 // Escuchar conexiones
46                 server_socket.listen(NULL, port, on_connect, 5);
47                 // NOT REACHED
48             }
49             catch (cpsm::Exception& e)
50             {
51                 e.dump();
52             }
53         }
54 };
55
56 int main(int argc, char* argv[])
57 {
58     if (argc != 2)
59     {
60         std::cout << "usage: " << argv[0] << " port" << std::endl;
61         return -1;
62     }
63
64     unsigned short port = atoi(argv[1]);
65
66     // Iniciar servidor
67     Server::startup(port);
68
69     // NOT REACHED
70
71     return 0;
```

Listado 3.34: Server1.cpp: programa servidor (primer intento)

En el Listado 3.34 puede apreciarse que, luego procesar los argumentos de línea de comando, se ejecuta el método `Server::startup()`. Dicho método inicia el servidor. Puntualmente, construye un `TCP Socket` y luego invoca el método `listen()` para aguardar conexiones. El tercer argumento de `listen()` es la función que debe ejecutarse al recibir una nueva conexión, en este caso `Server::on_connect()` (ver Listado 3.1). `on_connect()` simplemente realiza un `send()` y luego un `close()`⁶⁷ en el `socket` que representa la conexión, es decir, en el `socket` que recibe por parámetro.

3.4.2. Segundo intento: Comunicaciones y Concurrencia

El segundo intento es una modificación del programa servidor de la sección anterior, donde se utiliza un `thread` por conexión. El programa servidor es concurrente y para lograr esta funcionalidad se utiliza el servicio de Concurrencia del CPSM.

En la Figura 3.5 se muestra la secuencia de interacción entre el programa servidor y un cliente.

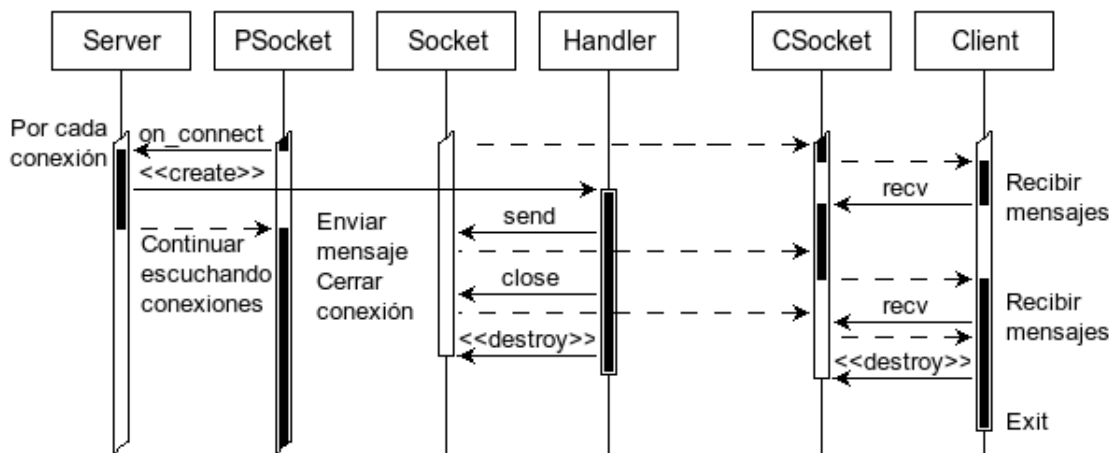


Figura 3.5: Secuencia de interacción para el segundo intento

La secuencia de interacción del segundo intento coincide, en lo que respecta al establecimiento de la conexión, con la presentada en la Figura 3.4 para el primer intento. Por este motivo, se omite en la Figura 3.5 dicha secuencia de establecimiento de la conexión. La principal diferencia con el intento anterior es la aparición de un nuevo objeto, `Handler`, que representa cada `thread` de ejecución creado para mantener una comunicación con un par conectado. Luego del evento `on_connect()`, el programa principal del servidor recupera brevemente el control del `thread` principal de la aplicación. Durante este lapso se crea un nuevo `thread` de ejecución (`Handler`) y se delega en él la comunicación con el cliente. En dicho `thread` se ejecuta una secuencia similar a la del intento anterior (`send()` y luego `close()`).

A continuación se muestra el código del programa servidor en este intento, denominado `Server2.cpp`:

⁶⁷La invocación a `close()` se realiza implícitamente al destruir el `socket`.

```
1 #include "cpsm/TCPsocket.hpp"
2 #include "cpsm/Thread.hpp"
3 #include "cpsm/Mutex.hpp"
4 #include "cpsm/Condition.hpp"
5
6 #include <iostream>
7 #include <list>
8
9 namespace cpsm = ce_cpsm;
10
11 /** Cada conexión es atendida por una instancia de Handler. Un Handler es una
12 * especialización de ThreadObject. Al iniciarse el Handler (start()), se
13 * ejecuta el método run() en otro thread.
14 *
15 * Handler provee, también, una ‘interfaz estática’ que permite administrar
16 * los Handlers creados y recolectar los threads completos.
17 *
18 * El Reaper Thread se inicia utilizando un ThreadVoid.
19 */
20 class Handler : public cpsm::ThreadObject
21 {
22     public:
23         /// Constructor: almacena el socket que representa la conexión
24         Handler(cpsm::TCPsocket* s) : _s(s) {}
25
26         /// Destructor: libera el socket que representa la conexión
27         virtual ~Handler() { delete _s; }
28
29         /// Función principal del Handler (se ejecuta en un thread propio)
30         virtual void run() throw ()
31         {
32             try
33             {
34                 std::cout << "handle peer" << std::endl;
35
36                 // Enviar mensaje
37                 const char msg[] = "\n\n*** Hello World! ***\n\n\n";
38                 _s->send(msg, sizeof(msg));
39
40                 std::cout << "handle peer done" << std::endl;
41             }
42             catch (cpsm::Exception& e)
43             {
44                 e.dump();
45             }
46
47             // notificar al Reaper Thread que este thread está por terminar
48             // y debe ser recolectado.
49             try
50             {
51                 cpsm::Mutex::Guard guard(_lock); // +---sección crítica-----+
52                 _done.push_back(this);          // |agregar thread a la lista|
53                                                 // |de threads completos.  |
54                 _cond.signal();                 // |notificar a Reaper Thread|
55                                                 // +-----+
56             }
57             catch (cpsm::Exception& e)
58             {
59                 e.dump();
60             }
61         }
62
63         /// Función principal del Reaper Thread (se ejecuta en un thread propio)
64         static void reaper(void* useless) throw ()
65         {
66             try
```

```
67     {
68         // --- sección crítica ----- //
69         cpsm::Mutex::Guard guard(_lock);
70
71         std::cout << "reaper: thread started" << std::endl;
72         for (;;) // ever
73         {
74             _cond.wait(); // Esperar la notificación de algún thread.
75                           // La espera libera _lock. Cuando wait()
76                           // retorna, _lock está bloqueado (sección
77                           // crítica).
78
79             std::cout << "reaper: got signal" << std::endl;
80
81             // para cada thread en la lista de threads completos...
82             done_t::iterator item;
83             for (item = _done.begin(); item != _done.end(); ++item)
84             {
85                 (*item)->join(); // esperar que el thread termine
86                 delete *item;    // liberar recursos del thread
87             }
88             std::cout << "reaper: " << _done.size() << " thread reaped"
89                       << std::endl;
90
91             _done.clear(); // vaciar la lista de threads completos
92         }
93         // --- fin sección crítica ----- //
94     }
95     catch (cpsm::Exception& e)
96     {
97         e.dump();
98     }
99 }
100
101 protected:
102     cpsm::TCPSocket* _s; ///< Socket de este Handler
103
104     ///< Declaración de atributos estáticos (Reaper Thread)
105     static cpsm::Mutex    _lock;    ///< Mutex de sección crítica
106     static cpsm::Condition _cond;    ///< Variable de notificación
107     typedef std::list<Handler*> done_t; ///< Tipo lista de threads completos
108     static done_t        _done;    ///< Lista de threads completos
109 };
110
111 // Definición de atributos estáticos (Reaper Thread)
112 cpsm::Mutex    Handler::_lock;
113 cpsm::Condition Handler::_cond(_lock);
114 Handler::done_t Handler::_done;
115
116 /** La clase Server agrupa el comportamiento y estado del servidor.
117  * El método principal de Server, startup(), inicia el servidor
118  * y escucha conexiones en el @a server_socket .
119  *
120  * El método on_connect() es invocado por cada conexión entrante. En él,
121  * se deriva la comunicación con el par conectado a un nuevo Handler.
122  */
123 class Server
124 {
125     public:
126         ///< Función callback invocada por cada conexión entrante
127         static bool on_connect(cpsm::TCPSocket* s) throw ()
128         {
129             Handler* h = new Handler(s); // Crear nuevo Handler
130             try
131             {
132                 std::cout << "on_connect" << std::endl;
```

```
133
134         h->start(); // Iniciar thread del Handler
135
136         std::cout << "on_connect done" << std::endl;
137
138         return true; // el socket fue utilizado
139     }
140     catch (cpsm::Exception& e)
141     {
142         e.dump();
143         delete h; // eliminar el Handler
144         return true; // el socket fue liberado por h.
145     }
146 }
147
148 /// Función principal del servidor
149 static void startup(unsigned short port) throw ()
150 {
151     // 1) Iniciar Reaper Thread (recolector de threads completos)
152     cpsm::ThreadVoid rip;
153     try
154     {
155         rip.start(Handler::reaper, NULL);
156     }
157     catch (cpsm::Exception& e)
158     {
159         e.dump();
160         return;
161     }
162
163     // 2) Crear socket pasivo y esperar conexiones
164     cpsm::TCPSocket server_socket;
165     try
166     {
167         // Escuchar conexiones
168         server_socket.listen(NULL, port, on_connect, 5);
169         // NOT REACHED
170     }
171     catch (cpsm::Exception& e)
172     {
173         e.dump();
174     }
175 }
176 };
177
178 int main(int argc, char* argv[])
179 {
180     if (argc != 2)
181     {
182         std::cout << "usage: " << argv[0] << " port" << std::endl;
183         return -1;
184     }
185
186     unsigned short port = atoi(argv[1]);
187
188     // Iniciar servidor
189     Server::startup(port);
190
191     // NOT REACHED
192
193     return 0;
194 }
```

Listado 3.35: Server2.cpp: programa servidor (segundo intento)

En el Listado 3.35 puede apreciarse que, tanto la función *main()*, cuanto la clase *Server*

son similares al intento anterior (ver Listado 3.34), excepto por la implementación del método estático *Server::on_connect()*, que en este intento, construye e inicia (*start()*) un *Handler* que se ocupará de atender la conexión en otro *thread*. Adicionalmente, se utiliza un *thread*, denominado *Reaper Thread*, que se encarga de recolectar todos los *threads* que finalizaron su ejecución y de destruir los recursos asociados a los mismos. La clase *Handler* es una especialización de *ThreadObject* (ver Listado 3.2) que, en su método *run()*, envía el mensaje “Hello World” (*send()*) y luego finaliza la conexión (*close()*). Antes de concluir su ejecución, *Handler::run()*, notifica al *Reaper Thread* su próxima finalización (*_cond.signal()*). Al recibir esta notificación, el *Reaper Thread* aguarda la finalización del *thread* (*join()*) y luego destruye los recursos asociados al mismo (*delete*).

3.4.3. Tercer intento: Comunicaciones, Concurrencia y Bibliotecas dinámicas

En el tercer intento se desacopla el protocolo de comunicaciones, de capa de aplicación, de los programas utilizados para establecer la conexión (*Server* y *Client*). Para llevar a cabo la separación se utiliza el servicio de Bibliotecas dinámicas provisto por el CPSM. Puntualmente, los programas servidor y cliente administran el establecimiento y la finalización de las conexiones, y delegan la interacción propia de la aplicación en un conjunto de funciones importadas de una biblioteca dinámica.

En la Figura 3.6 se muestra la secuencia de interacción entre el programa servidor y un cliente.

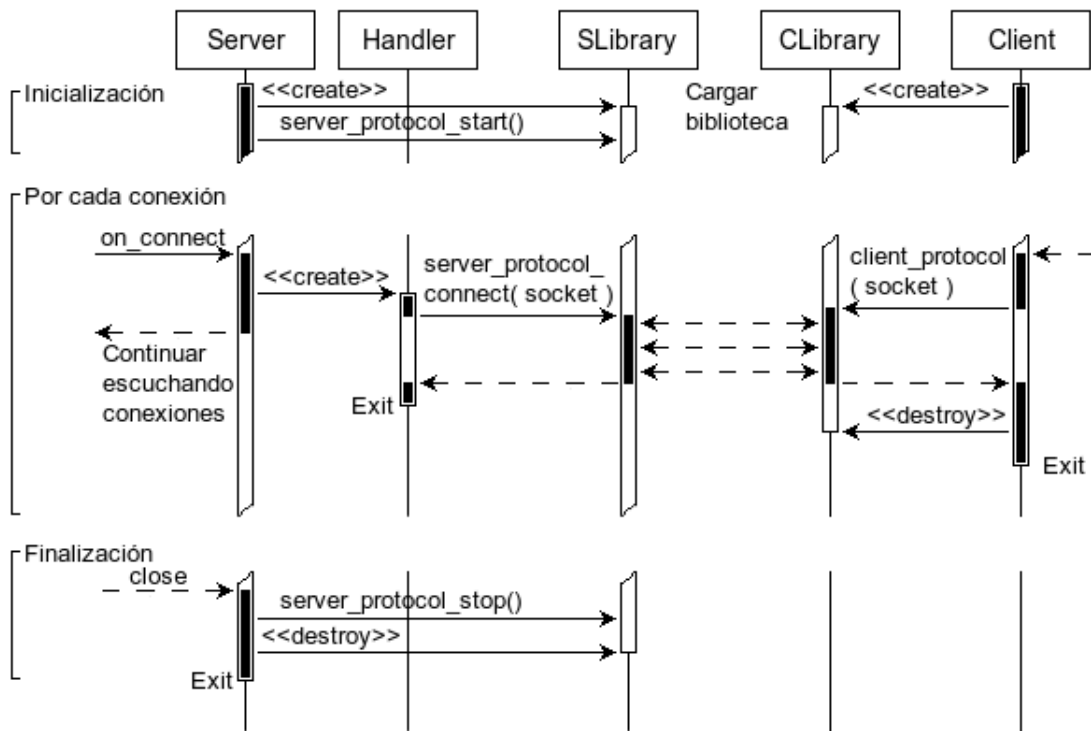


Figura 3.6: Secuencia de interacción para el tercer intento

Para simplificar la Figura 3.6 se omitieron todos los *sockets* utilizados durante la comunicación. Los eventos emitidos por el *socket* pasivo se representan como mensajes enviados a

Server. Estos eventos son: se estableció una nueva conexión (*on_connect*), y se cerró el *socket* pasivo (*close*).

En la Figura 3.6, los objetos interactuantes son, de izquierda a derecha:

- **Server**: la lógica principal de ejecución del programa servidor, representada por el comportamiento estático de la clase *Server*.
- **Handler**: instancia creada por cada conexión para mantener la comunicación con el par conectado en un *thread* secundario. En su constructor recibe el *socket* que representa la conexión establecida (indicado, en la Figura, como el argumento “*socket*” enviado a la función *server_protocol_connect()*).
- **SLibrary**: la instancia de *Library*, que representa la biblioteca dinámica asociada al servidor, en tiempo de ejecución.
- **CLibrary**: idem *SLibrary* en el cliente.
- **Cliente**: la lógica principal de ejecución del programa cliente.

La secuencia se presenta fragmentada en tres partes:

La primer parte corresponde al proceso de inicialización, donde el servidor asocia explícitamente la biblioteca dinámica e invoca una función de inicialización importada de dicha biblioteca. La inicialización en el cliente supone únicamente la asociación explícita de la biblioteca dinámica. En la Figura se agrupan las inicializaciones del cliente y el servidor en el primer fragmento, sin embargo, no es necesario que dichas inicializaciones produzcan en simultáneo. En rigor, el servidor se inicializa antes de comenzar a escuchar conexiones (*TCPSocket::listen()*), y el cliente antes de establecer la conexión (*TCPSocket::connect()*).

La segunda parte de la secuencia representa la interacción asociada con cada conexión. En el programa servidor, la secuencia se inicia con la indicación de establecimiento de una conexión (*on_connect()*), luego se crea un nuevo *thread* (*Handler*) para mantener la comunicación con el par conectado. El *thread* delega la comunicación en la función *server_protocol_connect()*, importada de la biblioteca. Dicha función es invocada con el *socket* que representa la conexión como argumento. Dada la naturaleza de las interacciones, la función *server_protocol_connect()* deberá ser *thread-safe* ya que puede ser invocada concurrentemente. El cliente, luego de establecer la conexión, delega la comunicación en la función *client_protocol()*, importada de la biblioteca, que es la contraparte de *server_protocol_connect()*. Al finalizar la comunicación, el *Handler* asignado a esa conexión se detiene.

La tercer parte corresponde a la etapa de finalización en la que el servidor invoca una función de finalización importada de la biblioteca y luego termina su ejecución.

El código correspondiente a las etapas de inicialización y finalización del servidor es:

```
int main(int argc, char* argv[])
{
    ...

    cpsm::Library library(library_name);

    Handler::handler_function = cpsm::findFunction<Handler::func_t>(library, "
server_protocol_connect");
    Server::start_function    = cpsm::findFunction<Server::start_func_t>(library, "
server_protocol_start");
    Server::stop_function     = cpsm::findFunction<Server::stop_func_t>(library, "
server_protocol_stop");
```

```
    Server::startup(port);
    ...
}

class Server
{
    ...
    static void startup(unsigned short port) throw ()
    {
        ...
        cpsm::TCPSocket server_socket;
        ...
        // Inicialización: server_protocol_start()
        (*start_function)(&server_socket);

        server_socket.listen(NULL, port, on_connect, 5);
        ...
        // Finalización: server_protocol_stop()
        (*stop_function)();
        ...
    }
    ...
};
```

Listado 3.36: Fragmento de las etapas de inicialización y finalización del servidor

El código correspondiente a la etapa de inicialización del cliente es:

```
int main(int argc, char* argv[])
{
    ...
    cpsm::Library library(library_name);

    typedef void (*func_t)(cpsm::TCPSocket* s);
    func_t handler_function = cpsm::findFunction<func_t>(library, "client_protocol"
    );
    ...
}
```

Listado 3.37: Fragmento de la etapa de inicialización del cliente

En ambos casos, las variables *library_name*, *port*, etc, se obtienen a de los argumentos de línea de comando a cada programa.

En la segunda parte de la secuencia de interacción, el servidor recibe la indicación de la conexión establecida (*on_connect()*) y crea e inicia un nuevo *thread* (*Handler*). En el nuevo *thread* delega la comunicación en la función *server_protocol_connect()* importada de la biblioteca dinámica asociada:

```
class Server
{
    ...
    static bool on_connect(cpsm::TCPSocket* s) throw ()
    {
        ...
        Handler* h = new Handler(s); // Crear nuevo Handler
        h->start(); // Iniciar thread del Handler
        ...
    }
    ...
};
```

```
class Handler
{
    ...
    virtual void run() throw ()
    {
        ...
        // server_protocol_connect()
        (*handler_function)(_s);
        ...
    }
    ...
};
```

Listado 3.38: Fragmento de la etapa de indicación de conexión en el servidor

El programa cliente, luego de establecer la conexión, delega la comunicación en la función *client_protocol()*, importada de la biblioteca dinámica asociada:

```
int main(int argc, char* argv[])
{
    ...
    cpsm::TCPSocket s;
    s.connect(address, port);

    // client_protocol()
    (*handler_function>(&s);
    ...
}
```

Listado 3.39: Fragmento de la etapa de establecimiento de la conexión en el cliente

Nótese que las variables *address*, *port*, etc., se obtienen de los argumentos de línea de comandos de la aplicación.

A continuación se muestra el código completo del programa servidor, denominado *Server3.cpp*:

```
1 #include "cpsm/TCPsocket.hpp"
2 #include "cpsm/Thread.hpp"
3 #include "cpsm/Mutex.hpp"
4 #include "cpsm/Condition.hpp"
5 #include "cpsm/Library.hpp"
6 #include <iostream>
7 #include <list>
8
9 namespace cpsm = ce_cpsm;
10
11 /** Cada conexión es atendida por una instancia de Handler. Un Handler es una
12  * especialización de ThreadObject. Al iniciarse el Handler (start()), se
13  * ejecuta el método run() en otro thread.
14  *
15  * Handler provee, también, una ‘interfaz estática’ que permite administrar
16  * los Handlers creados y recolectar los threads completos.
17  *
18  * El Reaper Thread se inicia utilizando un ThreadVoid y se finaliza a través
19  * del método Handler::reaper_stop().
20  */
21 class Handler : public cpsm::ThreadObject
22 {
23     public:
24         /// Prototipo de la función invocada por cada conexión (en otro thread)
25         typedef void (*func_t)(cpsm::TCPSocket* s);
26
27         /// Puntero estático a la función invocada por cada conexión
28         static func_t handler_function;
```

```
29
30     /// Constructor: almacena el socket que representa la conexión
31     Handler(cpsm::TCPSocket* s) : _s(s) {}
32
33     /// Destructor: libera el socket que representa la conexión
34     virtual ~Handler() { delete _s; }
35
36     /// Función principal del Handler (se ejecuta un thread propio)
37     virtual void run() throw ()
38     {
39         std::cout << "handle peer" << std::endl;
40
41         // server_protocol_connect()
42         (*handler_function)(_s);
43
44         std::cout << "handle peer done" << std::endl;
45
46         // notificar al Reaper Thread que este thread está por terminar
47         // y debe ser recolectado.
48         try
49         {
50             cpsm::Mutex::Guard guard(_lock); // +---sección crítica-----+
51             _done.push_back(this);           // |agregar thread a la lista|
52                                             // |de threads completos.   |
53             _cond.signal();                  // |notificar a Reaper Thread|
54                                             // +-----+
55         }
56         catch (cpsm::Exception& e)
57         {
58             e.dump();
59         }
60     }
61
62     /// Función principal del Reaper Thread (se ejecuta en un thread propio)
63     static void reaper(void* useless) throw ()
64     {
65         try
66         {
67             // --- sección crítica ----- //
68             cpsm::Mutex::Guard guard(_lock);
69
70             std::cout << "reaper: thread started" << std::endl;
71             for (;;) // ever
72             {
73                 _cond.wait(); // Esperar la notificación de algún thread.
74                             // La espera libera _lock. Cuando wait()
75                             // retorna, _lock está bloqueado (sección
76                             // crítica).
77
78                 std::cout << "reaper: got signal" << std::endl;
79
80                 // para cada thread en la lista de threads completos...
81                 done_t::iterator item;
82                 for (item = _done.begin(); item != _done.end(); ++item)
83                 {
84                     (*item)->join(); // esperar que el thread termine
85                     delete *item;    // liberar recursos del thread
86                 }
87                 std::cout << "reaper: " << _done.size() << " thread reaped"
88                     << std::endl;
89
90                 _done.clear(); // vaciar la lista de threads completos
91
92                 if (_stop_now) // si se debe detener el thread,
93                     break;    // abandonar el loop
94             }
95         }
96     }
97 }
```

```

95         // --- fin sección crítica ----- //
96     }
97     catch (cpsm::Exception& e)
98     {
99         e.dump();
100    }
101 }
102
103 /// Detener Reaper Thread
104 static void reaper_stop()
105 {
106     try
107     {
108         cpsm::Mutex::Guard guard(_lock); // +---sección crítica-----+
109         _stop_now = true;                // | detener thread en la      |
110                                         // | proxima iteración        |
111         _cond.signal();                  // | notificar thread        |
112                                         // +-----+
113     }
114     catch (cpsm::Exception& e)
115     {
116         e.dump();
117     }
118 }
119
120 protected:
121     cpsm::TCPSocket* _s; ///< Socket de este Handler
122
123     /// Declaración de atributos estáticos (Reaper Thread)
124     static cpsm::Mutex    _lock;        ///< Mutex de sección crítica
125     static cpsm::Condition _cond;      ///< Variable de notificación
126     typedef std::list<Handler*> done_t; ///< Tipo lista de threads completos
127     static done_t        _done;        ///< Lista de threads completos
128     static bool          _stop_now;    ///< Detener el thread?
129 };
130
131 // Definición de atributos estáticos (run)
132 Handler::func_t Handler::handler_function = NULL;
133
134 // Definición de atributos estáticos (Reaper Thread)
135 cpsm::Mutex    Handler::_lock;
136 cpsm::Condition Handler::_cond(_lock);
137 Handler::done_t Handler::_done;
138 bool          Handler::_stop_now = false;
139
140 /** La clase Server agrupa el comportamiento y estado del servidor.
141  * El método principal de Server, startup(), inicia el servidor
142  * y escucha conexiones en el @a server_socket .
143  *
144  * El método on_connect() es invocado por cada conexión entrante. En él,
145  * se deriva la comunicación con el par conectado a un nuevo Handler.
146  */
147 class Server
148 {
149     public:
150         /// Prototipo de la función de inicialización del servidor
151         typedef void (*start_func_t)(cpsm::TCPSocket* s);
152
153         /// Prototipo de la función de finalización del servidor
154         typedef void (*stop_func_t)();
155
156         /// Puntero estático a la función de inicialización
157         static start_func_t start_function;
158
159         /// Puntero estático a la función de finalización
160         static stop_func_t  stop_function;

```

```
161
162     /// Función callback invocada por cada la conexión entrante
163     static bool on_connect(cpsm::TCPSocket* s) throw ()
164     {
165         Handler* h = new Handler(s); // Crear nuevo Handler
166         try
167         {
168             std::cout << "on_connect" << std::endl;
169
170             h->start(); // Iniciar thread del Handler
171
172             std::cout << "on_connect done" << std::endl;
173
174             return true; // el socket fue utilizado
175         }
176         catch (cpsm::Exception& e)
177         {
178             e.dump();
179             delete h; // eliminar el Handler
180             return true; // el socket fue liberado por h.
181         }
182     }
183
184     /// Función principal del servidor
185     static void startup(unsigned short port) throw ()
186     {
187         // 1) Iniciar Reaper Thread (recolector de threads completos)
188         cpsm::ThreadVoid rip;
189         try
190         {
191             rip.start(Handler::reaper, NULL);
192         }
193         catch (cpsm::Exception& e)
194         {
195             e.dump();
196             return;
197         }
198
199         // 2) Inicializar protocolo, crear socket pasivo y esperar conexiones
200         cpsm::TCPSocket server_socket;
201         try
202         {
203             // Inicialización: server_protocol_start()
204             (*start_function)(&server_socket);
205
206             // Escuchar conexiones
207             server_socket.listen(NULL, port, on_connect, 5);
208         }
209         catch (cpsm::Exception& e)
210         {
211             e.dump();
212         }
213
214         // 3) Finalizar protocolo y detener al Reaper Thread
215         try
216         {
217             // Finalización: server_protocol_stop()
218             (*stop_function)();
219
220             Handler::reaper_stop(); // Detener Reaper Thread
221             rip.join(); // Esperar a que el thread termine
222         }
223         catch (cpsm::Exception& e)
224         {
225             e.dump();
226         }
227     }
228 }
```

```

227     }
228 };
229
230 // Definición de atributos estáticos (startup)
231 Server::start_func_t Server::start_function = NULL;
232 Server::stop_func_t Server::stop_function = NULL;
233
234 int main(int argc, char* argv[])
235 {
236     if (argc != 3)
237     {
238         std::cout << "usage: " << argv[0] << " port library" << std::endl;
239         return -1;
240     }
241
242     unsigned short port = atoi(argv[1]);
243     const char* library_name = argv[2];
244
245     try
246     {
247         // Asociar biblioteca dinámica
248         cpsm::Library library(library_name);
249
250         // Importar funciones de la biblioteca
251         Handler::handler_function = cpsm::findFunction<Handler::func_t>(library, "
server_protocol_connect");
252         Server::start_function = cpsm::findFunction<Server::start_func_t>(library, "
server_protocol_start");
253         Server::stop_function = cpsm::findFunction<Server::stop_func_t>(library, "
server_protocol_stop");
254
255         // Iniciar servidor
256         Server::startup(port);
257     }
258     catch (cpsm::Exception& e)
259     {
260         e.dump();
261         return -1;
262     }
263
264     return 0;
265 }

```

Listado 3.40: Server3.cpp: programa servidor (tercer intento)

Al igual que en el intento anterior (ver Listado 3.35), el programa servidor cuenta con un *thread* adicional encargado de recolectar los threads completos (*Reaper Thread*). Un detalle adicional del código presentado en el Listado 3.40 es que se provee un mecanismo para finalizar el *Reaper Thread* (a través del método estático *Handler::reaper_stop()*) ya que se asume que el protocolo, implementado en la biblioteca dinámica asociada, podría cerrar el *socket* pasivo pasado por parámetro a la función de inicialización, *server_protocol_start()*. Una vez cerrado el *socket* pasivo, el método *listen()* retornará el control del *thread* principal de la aplicación a *Server::startup()*⁶⁸ que podrá detener el *Reaper Thread* y luego terminar la ejecución del servidor.

A continuación se muestra el código completo del programa cliente, denominado *Client.cpp*:

```

1 #include "cpsm/TCPSocket.hpp"
2 #include "cpsm/Library.hpp"
3 #include <iostream>
4
5 namespace cpsm = ce_cpsm;

```

⁶⁸En la Figura 3.6 este evento se indica con el mensaje *close* recibido por *Server*.

```
6
7 int main(int argc, char* argv[])
8 {
9     if (argc != 4)
10    {
11        std::cout << "usage: " << argv[0] << " address port library" << std::endl;
12        return -1;
13    }
14
15    const char*    address      = argv[1];
16    unsigned short port        = atoi(argv[2]);
17    const char*    library_name = argv[3];
18
19    try
20    {
21        cpsm::Library library(library_name);
22
23        typedef void (*func_t)(cpsm::TCPSocket* s);
24        func_t handler_function = cpsm::findFunction<func_t>(library, "client_protocol"
25        );
26
27        cpsm::TCPSocket s;
28        s.connect(address, port);
29
30        // client_protocol()
31        (*handler_function>(&s);
32    }
33    catch (cpsm::Exception& e)
34    {
35        e.dump();
36        return -1;
37    }
38    return 0;
39 }
```

Listado 3.41: Client.cpp: programa cliente (tercer intento)

El cliente delega la comunicación en la función *client_protocol()* importada de la biblioteca dinámica asociada.

Con el objeto de completar el ejemplo, se presenta a continuación una biblioteca dinámica que exporta las funciones necesarias para utilizar ambos programas, a saber:

- *server_protocol_start()*
- *server_protocol_stop()*
- *server_protocol_connect()*
- *client_protocol()*

Esta biblioteca, en particular, provee un servicio de *broadcast* a todos los clientes conectados. Puntualmente, cada mensaje enviado por un cliente es reenviado a todos los demás. Los clientes envían los caracteres ingresados por *stdin* al servidor, mientras un thread secundario copia en *stdout* todos los caracteres recibidos del servidor.

Las funciones exportadas por la biblioteca son:

```
namespace cpsm = ce_cpsm;

LIBRARY_EXPORT void server_protocol_start(cpsm::TCPSocket* s) throw ();
LIBRARY_EXPORT void server_protocol_stop(cpsm::TCPSocket* s) throw ();
```



```
LIBRARY_EXPORT void server_protocol_connect(cpsm::TCPSocket* s) throw ();
LIBRARY_EXPORT void client_protocol(cpsm::TCPSocket* s) throw ();
```

Listado 3.42: Broadcast.cpp: fragmento de biblioteca dinámica

Donde cada función cumple el siguiente rol:

- *server_protocol_start()* rutina de inicialización del servidor, recibe el *socket* pasivo del servidor como argumento.
- *server_protocol_stop()* rutina de finalización del servidor.
- *server_protocol_connect()* rutina invocada concurrentemente al establecerse una conexión. Recibe un *socket* conectado como argumento.
- *client_protocol()* rutina de interacción con el servidor, ejecutada en el cliente. Es la contraparte de *server_protocol_connect()*.

client_protocol() utiliza dos *threads*. El *thread* principal de invocación lee caracteres de *stdin* y los envía por el *socket*. Mientras tanto, un *thread* secundario lee caracteres del *socket* y los escribe en *stdout*. A continuación se muestra el fragmento de la biblioteca que corresponde a este código:

```
/// Thread secundario: lee caracteres del socket y los escribe en stdout
static void client_protocol_echo(cpsm::TCPSocket* s)
{
    try
    {
        char buffer[256];
        unsigned long count;

        // recibir caracteres del socket...
        while ((count = s->recv(buffer, sizeof(buffer)-1)) != 0)
        {
            buffer[count] = '\0'; // completar la string
            std::cout << buffer; // mostrar por pantalla
        }
    }
    catch (cpsm::Exception& e)
    {
        e.dump();
    }
}

/** Una vez establecida la conexión, el cliente crea el thread secundario y,
 * en el thread de invocación, lee caracteres de stdin y los envía por el
 * socket. (Es line-buffered, es decir, envía de a una línea por vez).
 */
LIBRARY_EXPORT void client_protocol(cpsm::TCPSocket* s) throw ()
{
    try
    {
        std::cout << "client_protocol loaded" << std::endl;

        cpsm::Thread<cpsm::TCPSocket*> th; // Crear thread secundario
        th.start(client_protocol_echo, s); // Iniciar thread secundario

        std::string buffer;
        int i;
        for (;;) // ever
        {
```

```

    if ((i = getchar()) != EOF) // leer un caracter de stdin
    {
        char c = (char) i;
        buffer += c;
        if (c != '\n' && c != '\r')
            continue; // todavia no se leyó una línea
    }
    if (buffer.size()) // si hay algo, enviarlo por el socket
        s->send(buffer.c_str(), buffer.size());

    buffer = ""; // vaciar buffer

    if (i == EOF) // si no hay más input terminar el for
        break;
}

s->close(); // cerrar el socket
th.join(); // esperar al thread secundario

std::cout << "client_protocol unloaded" << std::endl;
}
catch (cpsm::Exception& e)
{
    e.dump();
}
}
}

```

Listado 3.43: Broadcast.cpp: cliente

El servidor utiliza un thread secundario (i.e. *Stop Event Thread*) que se bloquea intentando leer *stdin*. Al recibir un *char*, el *thread* secundario cierra el *socket* pasivo del servidor y todos los *sockets* de conexiones establecidas. El código del *Stop Event Thread* es:

```

class ServerHandler
{
    ...
protected:
    /// Declaración de atributos estáticos (Stop Event Thread)
    static cpsm::TCPSocket* _server_socket; ///< socket pasivo
    static cpsm::ThreadVoid _stop_event;   ///< thread secundario

    ...

    /// Función principal del thread secundario (Stop Event)
    static void _stop_event_thread(void* useless)
    {
        std::cout << "ENTER to stop" << std::endl;

        getchar(); // esperar que el usuario presione ENTER

        try
        {
            // --- sección crítica ----- //
            cpsm::Mutex::Guard guard(_mutex);
            std::cout << "protocol: stop!" << std::endl;

            _server_socket->close(); // cerrar socket principal del servidor

            // cerrar todas las conexiones establecidas
            sockets_t::iterator item;
            for (item = _sockets.begin(); item != _sockets.end(); ++item)
                (*item)->close();

            std::cout << "protocol: sockets closed" << std::endl;
            // --- fin sección crítica ----- //
        }
    }
}

```

```

    }
    catch (cpsm::Exception& e)
    {
        e.dump();
    }
}
};
/// Definición de atributos estáticos (Stop Event Thread)
cpsm::TCPSocket* ServerHandler::_server_socket = NULL; // se asigna en startup()
cpsm::ThreadVoid ServerHandler::_stop_event;

```

Listado 3.44: Broadcast.cpp: *Stop Event Thread*

Para implementar la funcionalidad de *broadcast*, el servidor debe mantener una lista de conexiones establecidas y proveer una función que difunda un mensaje a todos los clientes conectados. Estas funciones, así como las variables que mantienen el estado del servidor, se agrupan en la clase `ServerHandler`:

```

class ServerHandler
{
public:
    ...
    /// Agregar un socket a la lista de conexiones establecidas
    static void add(cpsm::TCPSocket* s)
    {
        cpsm::Mutex::Guard guard(_mutex); // +---sección crítica-----+
        _sockets.insert(s);              // | insertar socket          |
                                         // +-----+
    }

    /// Eliminar un socket de la lista de conexiones establecidas
    static void remove(cpsm::TCPSocket* s)
    {
        cpsm::Mutex::Guard guard(_mutex); // +---sección crítica-----+
        _sockets.erase(s);               // | eliminar socket          |
                                         // +-----+
    }

    /// Reenviar un mensaje a todos los sockets de la lista salvo a @p s .
    static void broadcast(cpsm::TCPSocket*s, const char* msg,
                          unsigned long size)
    {
        // --- sección crítica ----- //
        cpsm::Mutex::Guard guard(_mutex);
        sockets_t::iterator item;

        // para cada conexión establecida (item)...
        for (item = _sockets.begin(); item != _sockets.end(); ++item)
        {
            if (*item != s) // si *item no es el emisor,
                (*item)->send(msg, size); // reenviar el mensaje a *item
        }
        // --- fin sección crítica ----- //
    }

protected:
    ...
    /// Declaración de atributos estáticos (Broadcast)
    typedef std::set<cpsm::TCPSocket*> sockets_t; ///< tipo set de sockets
    static sockets_t _sockets;                  ///< "lista" de sockets
    static cpsm::Mutex _mutex;                  ///< mutex de sección crítica
    ...
};
...

```

```

// Definición de atributos estáticos (Broadcast)
ServerHandler::sockets_t ServerHandler::_sockets;
cpms::Mutex ServerHandler::_mutex;

```

Listado 3.45: Broadcast.cpp: mantenimiento de conexiones

Las funciones *add()* y *remove()* permiten mantener la lista de conexiones establecidas, mientras que la función *broadcast()* difunde un mensaje a todos los pares conectados al servidor exceptuando al emisor del mensaje. Al establecerse una conexión, se registra el *socket* en el *ServerHandler* y la función *server_protocol_connect()* comienza a recibir caracteres. Cada conjunto de caracteres recibido es difundido a todos los pares conectados. Al finalizarse la conexión, se elimina el *socket* de la lista de conexiones establecidas.

```

// Rutina ejecutada (concurrentemente) por cada conexión entrante
LIBRARY_EXPORT void server_protocol_connect(cpms::TCPSocket* s) throw ()
{
    ServerHandler::add(s); // registrar esta conexión establecida
    try
    {
        char buffer[256];
        unsigned long count;

        // recibir caracteres del socket...
        while ((count = s->recv(buffer, sizeof(buffer)-1)) != 0)
        {
            buffer[count] = '\0'; // completar la string
            std::cout << "[" << s << "]:" << buffer;

            // difundir el mensaje
            ServerHandler::broadcast(s, buffer, count);
        }
    }
    catch (cpms::Exception& e)
    {
        e.dump();
    }
    ServerHandler::remove(s); // eliminar esta conexión de la lista
}

```

Listado 3.46: Broadcast.cpp: proceso de una conexión

Durante la inicialización del servidor se asigna el *socket* pasivo en un atributo protegido de la clase *ServerHandler* (*_server_socket*) y se inicia el thread secundario (*Stop Event Thread*) que, al recibir un evento del usuario, cierra dicho *socket*. Esta acción ocasiona que el método *TCPSocket::listen()* retorne el control al método *Server::startup()* (ver Listado 3.36). Este método, a su vez, ejecuta la función de finalización del servidor (*server_protocol_stop()*). A continuación se muestra el código de inicialización y finalización del servidor:

```

class ServerHandler
{
public:
    // Evento de inicialización del servidor
    static void startup(cpms::TCPSocket* server_socket)
    {
        // almacenar el socket pasivo
        _server_socket = server_socket;

        // iniciar thread secundario (Stop Event)
        _stop_event.start(_stop_event_thread, NULL);
    }

    // Evento de finalización del servidor
    static void shutdown()

```

```

    {
        // esperar a que finalice el thread secundario (Stop Event)
        _stop_event.join();
    }
    ...
};
...
/// Inicializar el servidor
LIBRARY_EXPORT void server_protocol_start(cpsm::TCPSocket* s) throw ()
{
    try
    {
        ServerHandler::startup(s);
    }
    catch (cpsm::Exception& e)
    {
        e.dump();
    }
}

/// Finalizar el servidor
LIBRARY_EXPORT void server_protocol_stop(cpsm::TCPSocket* s) throw ()
{
    try
    {
        ServerHandler::shutdown();
    }
    catch (cpsm::Exception& e)
    {
        e.dump();
    }
}
...

```

Listado 3.47: Broadcast.cpp: inicialización y finalización del servidor

A continuación se muestra el código completo de la biblioteca de ejemplo detallada en los párrafos anteriores.

```

1 #include "defs.hpp"
2 #include "cpsm/TCPSocket.hpp"
3 #include "cpsm/Mutex.hpp"
4 #include "cpsm/Thread.hpp"
5
6 #include <iostream>
7 #include <set>
8
9 namespace cpsm = ce_cpsm;
10
11 // === Funciones exportadas ===== //
12 LIBRARY_EXPORT void server_protocol_start(cpsm::TCPSocket* s) throw ();
13 LIBRARY_EXPORT void server_protocol_stop() throw ();
14 LIBRARY_EXPORT void server_protocol_connect(cpsm::TCPSocket* s) throw ();
15 LIBRARY_EXPORT void client_protocol(cpsm::TCPSocket* s) throw ();
16
17
18 // === Servidor ===== //
19 /** La clase ServerHandler agrupa el comportamiento general del servidor.
20 * Posee métodos estáticos de inicialización (startup) y finalización
21 * (shutdown). ServerHandler mantiene una lista de sockets conectados a
22 * través de los métodos add y remove. El método broadcast envía un mensaje
23 * a todos los sockets en la lista salvo al socket emisor.
24 *
25 * El método _stop_event_thread se ejecuta en un thread separado y se bloquea
26 * esperando input del usuario (lee stdin). Cuando el usuario envía una línea
27 * de información por stdin (presiona ENTER en la consola), la función cierra

```

```
28 * el socket principal del server (socket pasivo) y todas las conexiones.
29 */
30 class ServerHandler
31 {
32     public:
33         /// Evento de inicialización del servidor
34         static void startup(cpsm::TCPSocket* server_socket)
35         {
36             // almacenar el socket pasivo
37             _server_socket = server_socket;
38
39             // iniciar thread secundario (Stop Event)
40             _stop_event.start(_stop_event_thread, NULL);
41         }
42
43         /// Evento de finalización del servidor
44         static void shutdown()
45         {
46             // esperar a que finalice el thread secundario (Stop Event)
47             _stop_event.join();
48         }
49
50         /// Agregar un socket a la lista de conexiones establecidas
51         static void add(cpsm::TCPSocket* s)
52         {
53             cpsm::Mutex::Guard guard(_mutex); // +---sección crítica-----+
54             _sockets.insert(s);              // | insertar socket          |
55                                             // +-----+
56         }
57
58         /// Eliminar un socket de la lista de conexiones establecidas
59         static void remove(cpsm::TCPSocket* s)
60         {
61             cpsm::Mutex::Guard guard(_mutex); // +---sección crítica-----+
62             _sockets.erase(s);                // | eliminar socket          |
63                                             // +-----+
64         }
65
66         /// Reenviar un mensaje a todos los sockets de la lista salvo a @p s .
67         static void broadcast(cpsm::TCPSocket*s, const char* msg,
68                               unsigned long size)
69         {
70             // --- sección crítica ----- //
71             cpsm::Mutex::Guard guard(_mutex);
72             sockets_t::iterator item;
73
74             // para cada conexión establecida (item)...
75             for (item = _sockets.begin(); item != _sockets.end(); ++item)
76             {
77                 if (*item != s) // si *item no es el emisor,
78                     (*item)->send(msg, size); // reenviar el mensaje a *item
79             }
80             // --- fin sección crítica ----- //
81         }
82
83     protected:
84         /// Declaración de atributos estáticos (Stop Event Thread)
85         static cpsm::TCPSocket* _server_socket; ///< socket pasivo
86         static cpsm::ThreadVoid _stop_event;   ///< thread secundario
87
88
89         /// Declaración de atributos estáticos (Broadcast)
90         typedef std::set<cpsm::TCPSocket*> sockets_t; ///< tipo set de sockets
91         static sockets_t _sockets;                 ///< 'lista' de sockets
92         static cpsm::Mutex _mutex;                 ///< mutex de sección crítica
93
```

```
94     /// Función principal del thread secundario (Stop Event)
95     static void _stop_event_thread(void* useless)
96     {
97         std::cout << "ENTER to stop" << std::endl;
98
99         getchar(); // esperar que el usuario presione ENTER
100
101         try
102         {
103             // --- sección crítica ----- //
104             cpsm::Mutex::Guard guard(_mutex);
105             std::cout << "protocol: stop!" << std::endl;
106
107             _server_socket->close(); // cerrar socket principal del servidor
108
109             // cerrar todas las conexiones establecidas
110             sockets_t::iterator item;
111             for (item = _sockets.begin(); item != _sockets.end(); ++item)
112                 (*item)->close();
113
114             std::cout << "protocol: sockets closed" << std::endl;
115             // --- fin sección crítica ----- //
116         }
117         catch (cpsm::Exception& e)
118         {
119             e.dump();
120         }
121     }
122 };
123 /// Definición de atributos estáticos (Stop Event Thread)
124 cpsm::TCPSocket* ServerHandler::_server_socket = NULL; // se asigna en startup()
125 cpsm::ThreadVoid ServerHandler::_stop_event;
126
127 /// Definición de atributos estáticos (Broadcast)
128 ServerHandler::sockets_t ServerHandler::_sockets;
129 cpsm::Mutex ServerHandler::_mutex;
130
131 /// Inicializar el servidor
132 LIBRARY_EXPORT void server_protocol_start(cpsm::TCPSocket* s) throw ()
133 {
134     try
135     {
136         ServerHandler::startup(s);
137     }
138     catch (cpsm::Exception& e)
139     {
140         e.dump();
141     }
142 }
143
144 /// Finalizar el servidor
145 LIBRARY_EXPORT void server_protocol_stop() throw ()
146 {
147     try
148     {
149         ServerHandler::shutdown();
150     }
151     catch (cpsm::Exception& e)
152     {
153         e.dump();
154     }
155 }
156
157 /// Rutina ejecutada (concurrentemente) por cada conexión entrante
158 LIBRARY_EXPORT void server_protocol_connect(cpsm::TCPSocket* s) throw ()
159 {
```

```
160     ServerHandler::add(s); // registrar esta conexión establecida
161     try
162     {
163         char buffer[256];
164         unsigned long count;
165
166         // recibir caracteres del socket...
167         while ((count = s->recv(buffer, sizeof(buffer)-1)) != 0)
168         {
169             buffer[count] = '\0'; // completar la string
170             std::cout << "[" << s << "]:" << buffer;
171
172             // difundir el mensaje
173             ServerHandler::broadcast(s, buffer, count);
174         }
175     }
176     catch (cpsm::Exception& e)
177     {
178         e.dump();
179     }
180     ServerHandler::remove(s); // eliminar esta conexión de la lista
181 }
182
183 // === Cliente ===== //
184 /// Thread secundario: lee caracteres del socket y los escribe en stdout
185 static void client_protocol_echo(cpsm::TCPSocket* s)
186 {
187     try
188     {
189         char buffer[256];
190         unsigned long count;
191
192         // recibir caracteres del socket...
193         while ((count = s->recv(buffer, sizeof(buffer)-1)) != 0)
194         {
195             buffer[count] = '\0'; // completar la string
196             std::cout << buffer; // mostrar por pantalla
197         }
198     }
199     catch (cpsm::Exception& e)
200     {
201         e.dump();
202     }
203 }
204
205 /** Una vez establecida la conexión, el cliente crea el thread secundario y,
206 * en el thread de invocación, lee caracteres de stdin y los envía por el
207 * socket. (Es line-buffered, es decir, envía de a una línea por vez).
208 */
209 LIBRARY_EXPORT void client_protocol(cpsm::TCPSocket* s) throw ()
210 {
211     try
212     {
213         std::cout << "client_protocol loaded" << std::endl;
214
215         cpsm::Thread<cpsm::TCPSocket*> th; // Crear thread secundario
216         th.start(client_protocol_echo, s); // Iniciar thread secundario
217
218         std::string buffer;
219         int i;
220         for (;;) // ever
221         {
222             if ((i = getchar()) != EOF) // leer un caracter de stdin
223             {
224                 char c = (char) i;
225                 buffer += c;
```



```
226         if (c != '\n' && c != '\r')
227             continue; // todavia no se leyó una linea
228     }
229     if (buffer.size()) // si hay algo, enviarlo por el socket
230         s->send(buffer.c_str(), buffer.size());
231
232     buffer = ""; // vaciar buffer
233
234     if (i == EOF) // si no hay mas input terminar el for
235         break;
236 }
237
238 s->close(); // cerrar el socket
239 th.join(); // esperar al thread secundario
240
241 std::cout << "client_protocol unloaded" << std::endl;
242 }
243 catch (cpsm::Exception& e)
244 {
245     e.dump();
246 }
247 }
```

Listado 3.48: Broadcast.cpp: biblioteca de servicio *broadcast*

4. Análisis y Comparación de CPSMs

En este capítulo se presentan varios CPSMs descriptos mediante los elementos del modelo formal (ver sección 1.1) y los parámetros característicos de un CPSM (ver sección 1.2). Se analizan características derivadas de las actividades de desarrollo de los mismos, entre ellas el paradigma (ver sección 2.1.1) y el lenguaje de programación (ver sección 2.1.2) utilizados, el propósito del CPSM (ver sección 2.1.3) y el mecanismo de abstracción (ver sección 2.4).

La descripción de los CPSM abarcará el análisis de los siguientes servicios: Concurrencia, Comunicaciones, Filesystem y Bibliotecas dinámicas, según la definición presentada en la sección 2.2.

4.1. Netscape Portable Runtime (NSPR)

El proyecto *Netscape Portable Runtime* (NSPR) es un CPSM desarrollado por la Mozilla Foundation y actualmente provee abstracción de la plataforma subyacente a proyectos como Mozilla Firefox, Mozilla Thunderbird y Mozilla SeaMonkey, entre otros.

El objetivo de NSPR es proveer un conjunto uniforme de servicios compatibles con una gran cantidad de plataformas (sistemas operativos). NSPR posee implementaciones en *userspace* para servicios que originalmente no se encuentran en las plataformas subyacentes. Adicionalmente, posee implementaciones alternativas para algunos servicios, que permiten aprovechar las mejores características de cada plataforma[Moz].

NSPR es compatible con las plataformas UNIX más populares (AIX, *BSD, Digital, GNU/Linux, HP-UX SGI IRIX, SCO, Solaris, etc), Windows, MacOS y BeOS[Moz98].

Entre los servicios provistos por NSPR se cuentan[His00, Moz08a]:

- **Concurrencia:** posee varias implementaciones de *Threads* (*kernel-threads*, cuando estén disponibles, y *user-threads*) y sincronización basada en *Monitors*. También provee *Mutex* y *Condition Variables*.
- **Comunicaciones:** servicio basado en *BSD Sockets* con soporte para los protocolos TCP/IP y UDP/IP.
- **Filesystem:** posee primitivas de manipulación de archivos y directorios, pero no encapsula los símbolos reservados del *filesystem* ni los delimitadores⁶⁹.
- **Bibliotecas dinámicas:** provee la asociación explícita de bibliotecas dinámicas, permitiendo cargar y descargar bibliotecas, y acceder a los símbolos definidos en las mismas.

Otros servicios provistos por NSPR son temporización, fecha y hora, manejo de memoria, memoria compartida, *hashtables* y listas, operaciones en 64-bits, etc[Moz08a].

NSPR está desarrollado en el lenguaje de programación C, y por lo tanto presenta un paradigma estructurado. Su propósito principal es el desarrollo de aplicaciones orientadas a las comunicaciones y sustenta los principales productos de Mozilla que pueden clasificarse dentro de esta categoría⁷⁰.

⁶⁹Los productos de Mozilla utilizan un modelo orientado a componentes sustentado sobre NSPR denominado XPCOM[Par01a]. XPCOM provee abstracción del *filesystem*, incluidos los símbolos delimitadores de directorios y archivos, entre otras cosas[Moz07a]

⁷⁰e.g. Mozilla Firefox, Mozilla Thunderbird, etc

El mecanismo de abstracción utilizado en NSPR es separación física de archivos y directorios, utilizando *GNU Autotools* (*Autoconf*, *Automake* y *Libtool*) para dirigir la compilación⁷¹. El archivo *configure.in* contiene la lógica de detección de la plataforma y determina qué archivos y directorios se deberán compilar. Como resultado del proceso de compilación de NSPR, se selecciona un archivo de definición de la plataforma y de la arquitectura y sobre la base de este archivo se genera el archivo *include/nspr/prcpufig.h*. Adicionalmente, se utiliza un único archivo para definir la interfaz mínima independiente de la plataforma que deberá implementarse en cada plataforma.

El Listado 4.1 muestra un fragmento del archivo único de definición de funciones independientes de la plataforma donde se describe el mecanismo utilizado para lograr la abstracción de la plataforma. En el Listado 4.2 se condensan fragmentos de varios archivos que ilustran un caso de abstracción, puntualmente la implementación de la función *socket()* para GNU/Linux y Microsoft Windows.

```

1 /* ... */
2
3 /*****
4 *****/ A Word about Model Dependent Function Naming Convention *****/
5 *****/
6
7 /*
8 NSPR 2.0 must implement its function across a range of platforms
9 including: MAC, Windows/16, Windows/95, Windows/NT, and several
10 variants of Unix. Each implementation shares common code as well
11 as having platform dependent portions. This standard describes how
12 the model dependent portions are to be implemented.
13
14 In header file pr/include/primpl.h, each publicly declared
15 platform dependent function is declared as:
16
17 NSPR_API void _PR_MD_FUNCTION( long arg1, long arg2 );
18 #define _PR_MD_FUNCTION _MD_FUNCTION
19
20 In header file pr/include/md/<platform>/<platform>.h,
21 each #define'd macro is redefined as one of:
22
23 #define _MD_FUNCTION <blanks>
24 #define _MD_FUNCTION <expanded macro>
25 #define _MD_FUNCTION <osFunction>
26 #define _MD_FUNCTION <_MD_Function>
27
28 Where:
29
30 <blanks> is no definition at all. In this case, the function is not implemented
31 and is never called for this platform.
32 For example:
33 #define _MD_INIT_CPUS()
34
35 <expanded macro> is a C language macro expansion.
36 For example:
37 #define _MD_CLEAN_THREAD(_thread) \
38     PR_BEGIN_MACRO \
39     PR_DestroyCondVar(_thread->md.asyncIOCVar); \
40     PR_DestroyLock(_thread->md.asyncIOLock); \
41     PR_END_MACRO
42
43 <osFunction> is some function implemented by the host operating system.
44 For example:

```

⁷¹En Microsoft Windows se utiliza el entorno de compilación MSYS que contiene una implementación de *GNU Autotools*, *GNU make*, etc.

```

45 #define _MD_EXIT          exit
46
47 <_MD_function> is the name of a function implemented for this platform in
48 pr/src/md/<platform>/<source>.c file.
49 For example:
50 #define      _MD_GETFILEINFO      _MD_GetFileInfo
51
52 In <source>.c, the implementation is:
53 PR_IMPLEMENT(PRInt32) _MD_GetFileInfo(const char *fn, PRFileInfo *info);
54 */
55
56 /* ... */

```

Listado 4.1: Fragmento de nsrpub/pr/include/private/primpl.h

Como puede apreciarse en el Listado 4.1, para cada primitiva se define, en este archivo, una *macro* denominada *_PR_MD_FUNCTION*, donde *FUNCTION* es el nombre de la primitiva en mayúscula, como equivalente a otra *macro* de nombre *_MD_FUNCTION*. Luego, para cada plataforma, se debe definir la *macro* *_MD_FUNCTION* según una de las opciones indicadas en el Listado 4.1.

```

1 /* Fragmento de nsrpub/pr/include/private/primpl.h
2 * Declaración de la función _PR_MD_SOCKET.
3 */
4 /* (...) */
5 extern PROsfd _PR_MD_SOCKET(int af, int type, int flags);
6 #define      _PR_MD_SOCKET _MD_SOCKET
7 /* (...) */
8
9 /* Fragmento de nsrpub/pr/src/io/prsocket.c
10 * Caso de uso de la función _PR_MD_SOCKET
11 */
12 /* (...) */
13 PR_IMPLEMENT(PRFileDesc*) PR_Socket(PRInt32 domain, PRInt32 type, PRInt32 proto)
14 {
15     PROsfd osfd;
16     PRFileDesc *fd;
17     /* (...) */
18
19     osfd = _PR_MD_SOCKET(domain, type, proto);
20     if (osfd == -1) {
21         return 0;
22     }
23     if (type == SOCK_STREAM)
24         fd = PR_AllocFileDesc(osfd, PR_GetTCPMethods());
25     else
26         fd = PR_AllocFileDesc(osfd, PR_GetUDPMethods());
27
28     /* (...) */
29
30     return fd;
31 }
32 /* (...) */
33
34 /* Fragmento de nsrpub/pr/include/md/_unixos.h
35 * Declaración de la función _MD_SOCKET (implementación en UNIX de
36 * _PR_MD_SOCKET)
37 */
38 /* (...) */
39 extern PRInt32      _MD_socket(int af, int type, int flags);
40 #define _MD_SOCKET _MD_socket
41 /* (...) */
42
43 /* Fragmento de nsrpub/pr/src/md/unix/unix.c
44 * Definición de la función _MD_socket (implementación en UNIX de

```

```

45 * _PR_MD_SOCKET)
46 */
47 /* (...) */
48 PRInt32 _MD_socket(PRInt32 domain, PRInt32 type, PRInt32 proto)
49 {
50     PRInt32 osfd, err;
51
52     osfd = socket(domain, type, proto);
53
54     /* (...) */
55
56     return(osfd);
57 }
58 /* (...) */
59
60 /* Fragmento de nsprpub/pr/src/md/windows/ntio.c
61 * Definición de la función _PR_MD_SOCKET (implementación en WINDOWS NT)
62 */
63 /* (...) */
64 PROsfd
65 _PR_MD_SOCKET(int af, int type, int flags)
66 {
67     SOCKET sock;
68
69     sock = socket(af, type, flags);
70
71     /* (...) */
72
73     return (PROsfd)sock;
74 }

```

Listado 4.2: Fragmentos de varios archivos relativos a la implementación de la función *socket()*.

En el Listado precedente se presenta un ejemplo de abstracción para la función de creación de un socket, denominada en NSPR, *_PR_MD_SOCKET()*. En el primer fragmento (correspondiente a *primpl.h*) se declara la función *_PR_MD_SOCKET()* y se define la *macro* *_PR_MD_SOCKET* con el valor *_MD_SOCKET*. En el segundo fragmento (correspondiente a *prsocket.c*) se muestra un caso de uso de la función *_PR_MD_SOCKET()* en una porción independiente de la plataforma del código fuente de NSPR. En el tercer listado (correspondiente a *_unixos.h*) se declara la función *_MD_socket()* que es la implementación concreta de *_PR_MD_SOCKET()* en plataformas UNIX. En el cuarto fragmento (correspondiente a *unix.c*) se define la función declarada en *_unixos.h*. En el último fragmento (correspondiente a *ntio.c*) se define la función *_PR_MD_SOCKET()* para Windows NT.

La siguiente tabla resume las características de NSPR:

Lenguaje	$s_0 = C$
Paradigma	Estructurado
Propósito	Desarrollo de programas orientados a las comunicaciones
Sistema de compilación	<i>GNU Autotools</i> y <i>MSYS</i>
Plataformas	$\mathcal{P} = \{UNIX, Windows, MacOS, BeOS\}$
Servicios	$\mathcal{S} = \{s_0, CONC, COMM, FILE, DL, \dots\}$
Mecanismo de abstracción	Separación física

4.2. ADAPTIVE Communication Environment (ACE)

El *ADAPTIVE Communication Environment* (ACE) es un CPSM desarrollado por un grupo de investigadores de la Universidad de Vanderbilt, la Universidad de Washington, St. Louis, y la Universidad de California, Irvine, liderados por Douglas C. Schmidt. El objetivo de ACE es simplificar el desarrollo de aplicaciones, orientadas a objetos, que utilicen servicios de Comunicaciones, Concurrencia, asociación explícita de Bibliotecas dinámicas, Comunicación entre procesos, *event demultiplexing*, etc. ACE se utiliza en proyectos de Ericsson, Bellcore, Siemens, Motorola, Kodak, y McDonnell Douglas. También es utilizado en numerosos proyectos académicos[Sch93].

Los autores de ACE lo describen como un *framework* compuesto por múltiples componentes que realizan actividades comunes en el software de comunicaciones[Sch06]. Dentro de la estructura de ACE, resulta de interés la capa de abstracción del sistema operativo (*ACE OS Adapter Layer*). Esta capa abstrae de la plataforma a todos los componentes de alto nivel de ACE. En este sentido, ACE podría considerarse un *framework* de alto nivel construido sobre un CPSM, siendo este CPSM la *ACE OS Adapter Layer*. Esta tesis se centra únicamente en ACE en cuanto a su capacidad de proveer servicios de forma independiente de la plataforma, para un conjunto de plataformas compatibles. Es decir, el único aspecto de ACE analizado es su capacidad de actuar como un CPSM. En el sitio de Douglas C. Schmidt, en la Universidad de Washington, St. Louis, pueden encontrarse numerosos *papers* que describen los componentes de alto nivel de ACE⁷². Adicionalmente, en [SSRB00] se describen algunos de los patrones de diseño utilizados en ACE.

ACE es compatible con Windows 32-bit (utilizando MSVC++, Borland C++ Builder, e IBM Visual Age), Mac OS X, múltiples versiones de UNIX (e.g. Solaris, SGI IRIX, DG/UX, HP-UX, AIX, UnixWare, SCO, Linux, *BSD), etc[Sch06].

Entre los servicios provistos por ACE se cuentan[Sch06]:

- **Concurrencia:** *Threads* y sincronización basada en *Mutex*, *Semaphore*, *Condition Variables* y *Events Objects*. Provee una implementación del patrón *scoped-locking*[Sch99a] para asegurar la liberación de *Mutexes* en secciones críticas.
- **Comunicaciones:** servicio basado en *BSD Sockets* con soporte para los protocolos TCP/IP y UDP/IP (*broadcast*, datagramas conectados, *multicast*). También posee interfaces construidas sobre la *Transport Layer Interface* (TLI)[Sch92]. Provee implementaciones del patrón *Acceptor-Connector*[SSRB00].
- **Filesystem:** posee primitivas de manipulación de archivos y directorios.
- **Bibliotecas dinámicas:** provee la asociación explícita de bibliotecas dinámicas, permitiendo cargar y descargar bibliotecas, y acceder a los símbolos definidos en las mismas.

Otros servicios provistos por ACE son Comunicación entre procesos, *event demultiplexing*, *signal handling*, administración de memoria compartida, (re)configuración dinámica de servicios (*Component Configurator*)[Sch06].

ACE está desarrollado en el lenguaje de programación C++, en el paradigma orientado a objetos.

El mecanismo de abstracción utilizado en ACE es compilación condicional. Para dirigir la compilación en UNIX se utiliza *GNU Autotools* (*Autoconf*, *Automake* y *Libtool*). En Microsoft

⁷²<http://www.cs.wustl.edu/~schmidt/ACE-papers.html>

Windows se utiliza el entorno *Microsoft Visual Studio (Microsoft Visual C++, MSVC)* para el cual se proveen proyectos de compilación. En los *scripts* de compilación (e.g. *configure.ac*) se detectan las características de la plataforma de compilación y se definen *macros* en consecuencia. La *ACE OS Adapter Layer* está compuesta por todos los archivos incluidos en el directorio *ace/*, cuyo nombre comienza con “OS” (e.g. *ace/OS_NS_Thread.h*). La organización de estos archivos es la siguiente: por cada servicio o primitiva abstracta existe un *header* que declara la interfaz (“.h”), y la implementación correspondiente (“.cpp”). Para algunos servicios se provee un archivo extra que contiene la declaración de funciones *inline* (“.inl”).

Por ejemplo, la abstracción de *Thread* en ACE (i.e. la clase *ACE_Thread*) utiliza la función *ACE_OS::thr_create()* de la *ACE OS Adapter Layer*. Esta función está declarada en el archivo *ace/OS_NS_Thread.h*. En el Listado 4.3 se muestra un fragmento de este archivo donde se declara la función *ACE_OS::thr_create()*.

```

1  /* ... */
2  /*
3   * Creates a new thread having @a flags attributes and running @a func
4   * with @a args (if @a thread_adapter is non-0 then @a func and @a args
5   * are ignored and are obtained from @a thread_adapter). @a thr_id
6   * and @a t_handle are set to the thread's ID and handle (?),
7   * respectively. The thread runs at @a priority priority (see
8   * below).
9   *
10  * The @a flags are a bitwise-OR of the following:
11  * = BEGIN<INDENT>
12  * THR_CANCEL_DISABLE, THR_CANCEL_ENABLE, THR_CANCEL_DEFERRED,
13  * THR_CANCEL_ASYNCHRONOUS, THR_BOUND, THR_NEW_LWP, THR_DETACHED,
14  * THR_SUSPENDED, THR_DAEMON, THR_JOINABLE, THR_SCHED_FIFO,
15  * THR_SCHED_RR, THR_SCHED_DEFAULT, THR_EXPLICIT_SCHED,
16  * THR_SCOPE_SYSTEM, THR_SCOPE_PROCESS
17  * = END<INDENT>
18  *
19  * By default, or if @a priority is set to
20  * ACE_DEFAULT_THREAD_PRIORITY, an "appropriate" priority value for
21  * the given scheduling policy (specified in @a flags, e.g.,
22  * @c THR_SCHED_DEFAULT) is used. This value is calculated
23  * dynamically, and is the median value between the minimum and
24  * maximum priority values for the given policy. If an explicit
25  * value is given, it is used. Note that actual priority values are
26  * EXTREMELY implementation-dependent, and are probably best
27  * avoided.
28  *
29  * Note that @a thread_adapter is always deleted by @c thr_create,
30  * therefore it must be allocated with global operator new.
31  *
32  * At the moment for @a thr_name a valid string is passed then this
33  * will be used on VxWorks to set the task name. If we just pass a pointer
34  * the name of the task is returned
35  */
36  extern ACE_Export
37  int thr_create (ACE_THR_FUNC func,
38                void *args,
39                long flags,
40                ACE_thread_t *thr_id,
41                ACE_hthread_t *t_handle = 0,
42                long priority = ACE_DEFAULT_THREAD_PRIORITY,
43                void *stack = 0,
44                size_t stacksize = ACE_DEFAULT_THREAD_STACKSIZE,
45                ACE_Base_Thread_Adapter *thread_adapter = 0,
46                const char** thr_name = 0);
47

```

48 `/* ... */`

Listado 4.3: Fragmento del archivo `ace/OS_NS_Thread.h`: declaración de la función `ACE_OS::thr_create()`.

Como consecuencia del mecanismo de compilación condicional, la definición de la función `ACE_OS::thr_create()`, en el archivo `ace/OS_NS_Thread.cpp` consta de aproximadamente 680 líneas de código fuente⁷³. La implementación de dicha función es excesivamente extensa y por lo tanto se omite. Se hace notar, sin embargo, que en su implementación se realizan aproximadamente 30 comparaciones de compilación condicional.

La siguiente tabla resume las características de ACE:

Lenguaje	$s_0 = \text{C++}$
Paradigma	Orientado a objetos
Propósito	Desarrollo de programas orientados a las comunicaciones
Sistema de compilación	<i>GNU Autotools</i> y MSVC
Plataformas	$\mathcal{P} = \{UNIX, Windows, MacOS\}$
Servicios	$\mathcal{S} = \{s_0, CONC, COMM, FILE, DL, \dots\}$
Mecanismo de abstracción	Compilación condicional

4.3. Simple DirectMedia Layer (SDL)

El proyecto *Simple DirectMedia Layer* (SDL) es un CPSM creado por Sam Lantinga. SDL es un CPSM orientado a proveer soporte Multimedia en diversas plataformas, entre ellas Windows 32-bit, Linux, MacOS, BeOS, *BSD, Solaris, IRIX y QNX[Lana]. También fue migrado a hardware específico como la consola de videojuegos Sony Playstation 2[Sla01].

Entre los servicios provistos por SDL se cuentan[Lanb]:

- **Concurrencia:** *Threads* y sincronización basada en *Mutex*, *Semaphore* y *Condition Variables*. Nota: la implementación de *Condition Variables* en SDL se realiza, de forma independiente de la plataforma, utilizando *Mutexes* y semáforos.
- **Bibliotecas dinámicas:** provee la asociación explícita de bibliotecas dinámicas, permitiendo cargar y descargar bibliotecas, y acceder a los símbolos definidos en las mismas. Este servicio está especializado para cargar bibliotecas dinámicas de OpenGL, aunque puede utilizarse para cargar cualquier tipo de biblioteca dinámica.

Otros servicios provistos por SDL son Video, *Window Management*, Eventos (cola de eventos similar a las asociadas a una ventana en los *toolkits* de interfaz gráfica), Joystick, Audio, CD-ROM, Timers, etc[Lan05].

SDL está desarrollado en el lenguaje de programación C, en el paradigma estructurado. Su propósito principal es el desarrollo de aplicaciones multimedia de forma independiente de la plataforma.

El mecanismo de abstracción utilizado en SDL es un mecanismo mixto, basado en separación física de archivos y directorios, pero que hace un uso frecuente de compilación condicional.

⁷³Resulta interesante comparar este número con el equivalente para otros CPSM. En SDL, por ejemplo, el código combinado de todas las implementaciones de la función de creación de un *thread*, `SDL_SYS_CreateThread`, no alcanza las 200 líneas de código fuente.

En ambientes UNIX, se utiliza *GNU Autotools* (*Autoconf*, *Automake* y *Libtool*) para dirigir la compilación. En Microsoft Windows se utiliza el entorno *Microsoft Visual Studio* (*Microsoft Visual C++*, MSVC) para el cual se proveen proyectos de compilación. Las interfaces abstractas se declaran en diversos archivos dentro del directorio *include*. La implementación concreta de los servicios se encuentra en un directorio por servicio, dentro de la carpeta *src*. Cada directorio contiene archivos con las implementaciones generales (independientes de la plataforma) y un subdirectorio por plataforma (con las implementaciones dependientes de la plataforma). En el directorio de cada servicio existe un archivo con el prefijo “*SDL_sys*”, seguido del nombre del servicio. Estos archivos definen la interfaz abstracta mínima que deberá implementarse en cada plataforma. También se incluye un archivo con el prefijo “*SDL_*”, luego el nombre del servicio y, por último, el sufijo “*_c.h*”. Este archivo determina qué implementación incluir en función de la plataforma *target*.

Por ejemplo, en el caso del servicio de Concurrencia, el código común a todas las plataformas se encuentra en el archivo *src/thread/SDL_thread.c*. El archivo *src/thread/SDL_systhread.h* declara las siguientes primitivas:

```

1
2 /* ... */
3
4 /* This function creates a thread, passing args to SDL_RunThread(),
5    saves a system-dependent thread id in thread->id, and returns 0
6    on success.
7 */
8 #ifdef SDL_PASSED_BEGINTHREAD_ENDTHREAD
9 extern int SDL_SYS_CreateThread(SDL_Thread *thread, void *args,
10                                pfnSDL_CurrentBeginThread pfnBeginThread, pfnSDL_CurrentEndThread pfnEndThread);
11 #else
12 extern int SDL_SYS_CreateThread(SDL_Thread *thread, void *args);
13 #endif
14
15 /* This function does any necessary setup in the child thread */
16 extern void SDL_SYS_SetupThread(void);
17
18 /* This function waits for the thread to finish and frees any data
19    allocated by SDL_SYS_CreateThread()
20 */
21 extern void SDL_SYS_WaitThread(SDL_Thread *thread);
22
23 /* This function kills the thread and returns */
24 extern void SDL_SYS_KillThread(SDL_Thread *thread);
25 /* ... */

```

Listado 4.4: Fragmento de *src/thread/SDL_systhread.h*: declaración de primitivas

A continuación se muestra un fragmento del archivo que selecciona la implementación dependiente de la plataforma, *src/thread/SDL_thread_c.h*:

```

1 /* ... */
2 /* Need the definitions of SYS_ThreadHandle */
3 #if SDL_THREADS_DISABLED
4 #include "generic/SDL_systhread_c.h"
5 #elif SDL_THREAD_BEOS
6 #include "beos/SDL_systhread_c.h"
7 #elif SDL_THREAD_DC
8 #include "dc/SDL_systhread_c.h"
9 #elif SDL_THREAD_OS2
10 #include "os2/SDL_systhread_c.h"
11 #elif SDL_THREAD_PTH
12 #include "pth/SDL_systhread_c.h"
13 #elif SDL_THREAD_PTHREAD

```

```

14 #include "pthread/SDL_systhread.c.h"
15 #elif SDL_THREAD_SPROC
16 #include "irix/SDL_systhread.c.h"
17 #elif SDL_THREAD_WIN32
18 #include "win32/SDL_systhread.c.h"
19 #elif SDL_THREAD_SYMBIAN
20 #include "symbian/SDL_systhread.c.h"
21 #else
22 #error Need thread implementation for this platform
23 #include "generic/SDL_systhread.c.h"
24 #endif
25 #include "../SDL_error.c.h"
26
27 /* This is the system-independent thread info structure */
28 struct SDL_Thread {
29     Uint32 threadid;
30     SYS_ThreadHandle handle;
31     int status;
32     SDL_error errbuf;
33     void *data;
34 };
35
36 /* This is the function called to run a thread */
37 extern void SDL_RunThread(void *data);
38
39 /* ... */

```

Listado 4.5: Fragmento de src/thread/SDL_thread.c.h: selección de implementación

Nótese que, en el Listado 4.5, la definición de las *macros* que determinan la plataforma *target* (e.g. *SDL_THREAD_WIN32*, *SDL_THREAD_PTHREAD*, etc) surge del *script* de compilación utilizado (i.e. *configure.in*).

La siguiente tabla resume las características de SDL:

Lenguaje	$s_0 = C$
Paradigma	Estructurado
Propósito	Desarrollo de programas multimedia
Sistema de compilación	<i>GNU Autotools</i> y <i>MSVC</i>
Plataformas	$\mathcal{P} = \{UNIX, Windows, MacOS, BeOS, \dots\}$
Servicios	$\mathcal{S} = \{s_0, CONC, DL, \dots\}$
Mecanismo de abstracción	Separación física y Compilación condicional

4.4. Boost

El proyecto *Boost* es un conjunto de bibliotecas escritas en C++ aportadas por una gran comunidad de programadores y empresas. Cada biblioteca de Boost provee un servicio específico, por ejemplo *Boost.Thread* provee el servicio de *Concurrencia*, mientras que *Boost.Interprocess* provee Comunicación entre procesos. En conjunto, las bibliotecas Boost constituyen un CPSM de gran *Flexibilidad*. En la versión actual (1.35), Boost provee aproximadamente 87 bibliotecas[DAR08] y es utilizado por Adobe, SAP, McAfee y Google, entre otros[ea08]. Varias bibliotecas de Boost ya forman parte del *C++ Technical Report 1*[JTC03]. Adicionalmente, algunas bibliotecas Boost, como *Boost.Filesystem*, fueron aceptadas para el *C++ Technical Report 2*[Daw08a].

Cada biblioteca de Boost posee un conjunto de plataformas compatibles propio, sin embargo todas son compatibles con Microsoft Windows y UNIX[Daw08b].

Entre los servicios provistos por Boost se cuentan[DAR08]:

- **Concurrencia:** *Threads* y sincronización basada en *Mutex*, *Condition Variables* y *Barriers*. Este servicio se implementa en la biblioteca *Boost.Thread*[Kem06].
- **Comunicaciones:** soporte para los protocolos TCP/IP y UDP/IP mediante una implementación modificada del patrón *Acceptor-Connector*[Sch97]. Soporta IPv4 e IPv6, también provee una implementación de SSL. Este servicio se implementa en la biblioteca *Boost.ASIO*[Koh08].
- **Filesystem:** posee primitivas de manipulación de archivos y directorios, iteración de directorios y abstracción de los símbolos reservados del filesystem (delimitadores, etc). Este servicio se implementa en la biblioteca *Boost.Filesystem*[Daw08a].

Otros servicios provistos por Boost incluyen CRC (código de redundancia cíclica), fecha y hora, funciones de *hash*, *Message Passing Interface*, metaprogramación, funciones matemáticas, *memory pools*, expresiones regulares, etc[DAR08].

Boost utiliza un programa de compilación propio denominado *Boost JAM* para dirigir la compilación. Las interfaces abstractas de cada servicio (biblioteca) se declaran en un directorio por servicio, dentro del directorio *boost* y su implementación concreta se encuentra en un directorio por servicio, dentro de la carpeta *libs*.

Puesto que, en Boost, cada servicio se provee a través de una biblioteca independiente, los mecanismos de abstracción varían según la biblioteca. En algunos casos, se utiliza compilación condicional, más aún, algunas bibliotecas de Boost están completamente implementadas en *headers* (i.e. no generan archivos de código objeto propios), un ejemplo de este caso es *Boost.Interprocess*. Por otro lado, algunas bibliotecas utilizan el mecanismo de separación física de archivos y directorios, por ejemplo *Boost.Thread*, que posee un subdirectorio por plataforma dentro del directorio *libs/thread/src*.

La siguiente tabla resume las características de Boost:

Lenguaje	$s_0 = C++$
Paradigma	Orientado a objetos
Propósito	General
Sistema de compilación	<i>Boost JAM</i>
Plataformas	$\mathcal{P} = \{UNIX, Windows, \dots(\text{depende del servicio})\}$
Servicios	$\mathcal{S} = \{s_0, CONC, COMM, FILE, \dots\}$
Mecanismo de abstracción	(depende del servicio)

4.5. wxWidgets

El proyecto wxWidgets es un CPSM desarrollado por Julian Smart en la Universidad de Edimburgo y actualmente es utilizado por AMD, AOL, NASA, Red Hat, W3C (Amaya) y Xerox, entre otros[SRZea08c].

wxWidgets es un CPSM orientado a proveer soporte para interfaces gráficas de usuario (*graphical user interfaces*, GUIs) utilizando el *look-and-feel* propio de la plataforma subyacente [SRZea08a]. wxWidgets es compatible con Microsoft Windows, GNU/Linux y UNIX (utilizando GTK+, Motif o el X Window System), y MacOS[SRZea08b].

Entre los servicios provistos por wxWidgets se cuentan[SRZea08b, SRZea07]:

- **Concurrencia:** *Threads* y sincronización basada en *Mutex*, *Semaphore* y *Condition Variables*.
- **Comunicaciones:** soporte para los protocolos TCP/IP y UDP/IP mediante una implementación modificada del patrón *Acceptor-Connector* [Sch97] (modelizado a través de las clases *wxSocketServer* y *wxSocketClient*, respectivamente).
- **Filesystem:** posee primitivas de manipulación de archivos y directorios, en particular, las clases *wxFilesystem* y *wxFileName* proveen iteración de directorios y abstracción de los símbolos reservados del filesystem (delimitadores, etc).
- **Bibliotecas dinámicas:** provee la asociación explícita de bibliotecas dinámicas, permitiendo cargar y descargar bibliotecas, y acceder a los símbolos definidos en las mismas a través de la clase *wxDynamicLibrary*.

Otros servicios provistos por wxWidgets son GUI (*document/view*, *drag-and-drop*, etc), conexión con bases de datos (ODBC), integración con OpenGL, implementación de protocolos de capa de aplicación (HTTP, FTP), soporte XML y HTML, compresión (mediante *zlib*), etc [SRZea08b, SRZea07].

wxWidgets está desarrollado en el lenguaje de programación C++, en el paradigma orientado a objetos. Su propósito principal es el desarrollo de aplicaciones con GUIs de forma independiente de la plataforma, pero con el *look-and-feel* propio de cada plataforma.

El mecanismo de abstracción utilizado en wxWidgets es separación física de archivos y directorios, utilizando *GNU Autotools* (*Autoconf*, *Automake* y *Libtool*) para dirigir la compilación en ambientes UNIX. En Microsoft Windows se utiliza el entorno *Microsoft Visual Studio* (*Microsoft Visual C++*, MSVC) para el cual se proveen proyectos *template* que sirven de punto de partida para el desarrollo. Las interfaces abstractas se declaran en diversos archivos dentro del directorio *include* y su implementación concreta se encuentra en un directorio por plataforma, dentro de la carpeta *src*. En el caso del servicio de Comunicaciones, el código común a todas las plataformas se encuentra en los archivos *socket.h/cpp* y su implementación delega en la clase *GSocket* (en los archivos *gsocket.h/cpp*) la abstracción del servicio. En el caso del servicio de Concurrencia, la interfaz abstracta se declara en *include/thread.h* pero cada plataforma posee una implementación completamente independiente.

La siguiente tabla resume las características de wxWidgets:

Lenguaje	$s_0 = \text{C++}$
Paradigma	Orientado a objetos
Propósito	Desarrollo de programas con interfaz gráfica de usuario
Sistema de compilación	<i>GNU Autotools</i> y MSVC
Plataformas	$\mathcal{P} = \{UNIX, Windows, MacOS\}$
Servicios	$\mathcal{S} = \{s_0, CONC, COMM, FILE, DL, \dots\}$
Mecanismo de abstracción	Separación física

4.6. Resumen

Las siguientes tablas resumen las características de los CPSMs descritos en este capítulo.

Características generales

CPSM	Lenguaje	Paradigma	Propósito	Sistema de compilación
NSPR	C	Estructurado	Comunicaciones	<i>GNU Autotools</i> y MSYS
ACE	C++	O. a objetos	Comunicaciones	<i>GNU Autotools</i> y MSVC
SDL	C	Estructurado	Multimedia	<i>GNU Autotools</i> y MSVC
Boost	C++	O. a objetos	General	<i>Boost JAM</i>
wxWidgets	C++	O. a objetos	GUI	<i>GNU Autotools</i> y MSVC

Servicios

CPSM	Concurrencia	Comunicaciones	Filesystem	Bibliotecas dinámicas
NSPR	SI	SI	SI	SI
ACE	SI	SI	SI	SI
SDL	SI	NO	NO	SI
Boost	SI	SI	SI	NO
wxWidgets	SI	SI	SI	SI

Plataformas y Mecanismo de abstracción

CPSM	Plataformas	Mecanismo de abstracción
NSPR	UNIX, Windows, MacOS, ...	Separación física
ACE	UNIX, Windows, MacOS, ...	Compilación condicional
SDL	UNIX, Windows, MacOS, ...	Separación física y Compilación condicional
Boost	UNIX, Windows, ...(depende del servicio)	(depende del servicio)
wxWidgets	UNIX, Windows, MacOS	Separación física

5. Conclusiones y futuras líneas de estudio

La literatura técnica sobre desarrollo de *software* independiente de la plataforma suele centrarse en uno de dos aspectos: cómo dotar a un programa de independencia de la plataforma, para un conjunto dado de plataformas; o cómo utilizar un determinado desarrollo (i.e. *framework*, API, biblioteca) para lograr independencia de la plataforma.

Respecto al desarrollo de CPSMs, los textos centrados en el primer aspecto resultan marginalmente útiles, ya que únicamente proveen una solución posible a un problema concreto, por ejemplo la implementación de *Condition Variables* en Microsoft Windows[SP]. Los textos centrados en el segundo aspecto aportan información acerca de los tipos de servicios provistos y las plataformas compatibles, para algunos CPSMs existentes. Esta información es valiosa en tanto y en cuanto muestra casos empíricamente viables de construcción de CPSMs. Sin embargo, tanto los textos centrados en el primer aspecto, cuanto los centrados en el segundo, se abstienen de profundizar en el proceso de desarrollo de un CPSM. Por lo tanto, un estudio de dicho proceso demanda una tarea de reconstrucción a partir de fragmentos de información.

Dentro de los principales desafíos presentados por el análisis realizado en los capítulos precedentes puede mencionarse:

- Reconstruir, a partir de ejemplos completos de CPSMs, las actividades y decisiones involucradas en el desarrollo de un CPSM.
- Determinar la interacción de los diferentes aspectos de un CPSM, por ejemplo la relación entre parámetros presentada en la sección 1.2.6 o la comparación de características predominantes del CPSM, según su propósito, presentada en la sección 2.1.3.1.

Para enfrentar los desafíos indicados, se construyó un conjunto de herramientas que permiten analizar el proceso de desarrollo de un CPSM y comparar distintos CPSMs. A través del Modelo formal (sección 1.1) y los parámetros característicos (sección 1.2) se puede describir la estructura de los CPSMs y se los puede clasificar en relación a sus parámetros predominantes.

Se analizaron las principales actividades involucradas en el desarrollo de un CPSM, entre ellas la selección de plataformas con las que es compatible (sección 2.3) y los servicios que provee (sección 2.2). También se discutieron dos mecanismos de abstracción que permiten aislar a las capas superiores (i.e. programas) de los detalles de la plataforma subyacente: compilación condicional (sección 2.4.2.1) y separación física de archivos y directorios (sección 2.4.2.2).

Respecto de la selección de implementaciones alternativas de un servicio, se indicaron algunos patrones de diseño que pueden utilizarse y se presentó un ejemplo de aplicación del patrón *Factory Method*[GHJV95] (sección 2.4.3).

Se reconocieron dos esquemas de inicialización y finalización de servicios: el implícito y el explícito (sección 2.4.4). Cada esquema presenta ventajas y desventajas. Un patrón que puede, bajo ciertas restricciones, simplificar la implementación de un esquema de inicialización y finalización implícita en un CPSM consiste en utilizar una instancia estática de una clase *Initializer*. Este patrón se fue descrito en la sección 2.4.4.

Las actividades y decisiones involucradas en el desarrollo de un CPSM se ilustraron en un Caso de Estudio completo (Capítulo 3) que provee los servicios de Concurrencia, Comunicaciones y asociación explícita de Bibliotecas dinámicas, compatible con Microsoft Windows y GNU/Linux. En particular, se realizó un especificación de las interfaces de los servicios provistos, utilizando el sistema de documentación *Doxygen* (sección 3.2.1), y su respectiva im-

plementación en las plataformas compatibles con el CPSM (sección 3.3). También se presentó, como ejemplo, un programa sustentado sobre el CPSM del Caso de Estudio (sección 3.4).

Adicionalmente, se realizó una descripción de algunos CPSMs *open-source* de características diversas, lo que permitió ilustrar decisiones y actividades involucradas en el desarrollo de un CPSM (Capítulo 4). La comparación de dichos CPSMs (en la sección 4.6) constituye un buen punto de partida para seleccionar un CPSM sobre el cual sustentar una aplicación.

El análisis realizado en este trabajo permite caracterizar un tipo de *software* muy particular: el CPSM. A través de las herramientas de análisis presentadas, es posible comparar CPSMs en relación a diferentes aspectos, por ejemplo: sus parámetros, el mecanismo de abstracción utilizado, las soluciones que plantean a determinados problemas de implementación, etc.

Por otra parte, pueden mencionarse varias líneas de investigación futura en relación al desarrollo de CPSMs:

- **Alternativas a un CPSM:** existen diversas alternativas a un CPSM para desarrollar *software* independiente de la plataforma. Algunas de ellas son:
 - Emulación del entorno de ejecución (e.g. Wine, Cygwin), es decir, ejecutar programas compilados para una plataforma dentro de un emulador de entorno, que se ejecuta en otra plataforma. Por ejemplo, el programa Wine permite ejecutar programas binarios en formato *Portable Executable*⁷⁴ en plataformas GNU/Linux.
 - Programas interpretados (e.g. Python, Perl, PHP, etc). Resulta de particular interés la manera en la que están desarrollados los intérpretes, ya que éstos enfrentan un problema similar al que los CPSMs deben resolver: abstraer servicios de la plataforma subyacente. En consecuencia, dichos intérpretes podrían sustentarse sobre un CPSM.
 - Un caso similar al anterior es la compilación a formas binarias independientes de la plataforma (*bytecode*), que luego son interpretadas por una máquina virtual (e.g. Java).
- **Pruebas y depuración (*debug*):** la naturaleza multiplataforma de un CPSM impacta sobre el desarrollo de pruebas de comportamiento y sobre la detección y corrección de errores en el CPSM. Puntualmente, todos y cada uno de los servicios provistos por un CPSM deben exhibir el mismo comportamiento en todas las plataformas con las que es compatible. Asegurar el comportamiento homogéneo del CPSM en todas las plataformas compatibles dista de ser trivial. Resulta de particular interés el desarrollo de pruebas lo suficientemente exhaustivas para asegurar este comportamiento.
- **“MetaCPSM”:** en términos del Modelo formal, podría definirse un MetaCPSM como un CPSM cuyo conjunto de plataformas compatibles (\mathcal{P}) esté integrado por CPSMs. De esta forma, el MetaCPSM podría utilizar, en cada plataforma, una implementación basada en un CPSM distinto, que provea mejor soporte en dicha plataforma. Resulta de interés analizar el impacto sobre la *Performance* de un MetaCPSM en relación a la de un CPSM. Puntualmente, estudiar si el aumento en la *Performance* que se adquiere al seleccionar la implementación más “adecuada” es significativo en relación a la degradación de la *Performance* asociada a un nivel de indirección adicional.

⁷⁴Un formato ejecutable binario utilizado en Microsoft Windows.

- **Extensión del Modelo Formal:** el Modelo formal, como herramienta de análisis, no se restringe a los CPSMs. Podría utilizarse para analizar otros casos, como por ejemplo la construcción de compiladores para un nuevo lenguaje de programación (en la sección 1.1 se presentó un breve ejemplo de este caso). Una extensión del modelo formal podría eliminar la *condición de existencia del punto de articulación* para contemplar casos como la compilación de un lenguaje a código de máquina para diferentes arquitecturas de *hardware*.

Apéndice

A. Glosario

A lo largo de esta Tesis se utilizan varios conceptos cuyo significado puede interpretarse de diversas maneras según el contexto en que se encuentren. El objetivo de este Anexo es restar ambigüedad a algunos términos clave utilizados en el análisis de CPSMs.

- **Expresividad:** en relación a una interfaz, representa la diversidad de operaciones que es posible realizar a través de dicha interfaz. Por ejemplo la expresividad de una interfaz cuya única primitiva sea *int sumar2mas3()* es menor que la de otra interfaz cuya única primitiva sea *int sumar(int a, int b)*, dado que la segunda permite sumar cualquier par de números *a* y *b*, mientras que la primera sólo es capaz de calcular la suma entre 2 y 3.
- **Implementación:** La implementación de una interfaz es el código concreto que cumple con el contrato que representa la interfaz de un servicio, asegurando que todas las operaciones primitivas funcionan según su especificación, cumpliendo con las características tanto sintácticas cuanto semánticas de cada una [GHJV95, p. 22]. En la sección 3.3 se presentan varios ejemplos de implementación de interfaces abstractas de los servicios provistos por un CPSM.
- **Interfaz:** La interfaz asociada a un servicio es una definición completa de las operaciones primitivas que constituyen dicho servicio, tanto sintáctica como semánticamente. Dentro de las características sintácticas de las primitivas, se puede mencionar tipo y número de los argumentos, tipo del valor de retorno, posibles excepciones y condiciones de error [Obj]. Las características semánticas incluyen interpretación de los argumentos, el valor de retorno y las excepciones, pre y postcondiciones, orden de invocación, etc [Obj04]. Las interfaces de los servicios provistos por una plataforma se denominan interfaces nativas. Las interfaces, independientes de la plataforma, de los servicios provistos por un CPSM se denominan interfaces abstractas.
- **Interfaz abstracta:** la interfaz de un servicio, especificada en forma independiente de la plataforma. Surge como abstracción del conjunto de interfaces nativas de diferentes plataformas. En las secciones 2.2.1 y 3.2.1 se dan varios ejemplos de interfaces abstractas de servicios provistos por un CPSM.
- **Interfaz nativa:** la interfaz de un servicio, presente en una plataforma específica. Se contrapone al concepto de interfaz abstracta. Un ejemplo de interfaz nativa es la interfaz del servicio de Comunicaciones en Microsoft Windows (*Winsock*).
- **Middleware:** capa de abstracción cuyo objetivo es encapsular los detalles de las interfaces nativas provistas por distintas plataformas, en una única interfaz homogénea. El término se utiliza en un sentido amplio, denotando una capa de abstracción sobre un conjunto de plataformas, sin restringir su alcance a los sistemas operativos de red, ni su aplicación a los sistemas distribuidos [Tan02].
- **Plataforma:** el conjunto de tecnologías que soporta la ejecución de programas. Puntualmente, un sistema operativo y la arquitectura de *hardware* que lo sustenta. En esta

tesis se utiliza el término “plataforma” como referencia a un sistema operativo, sin embargo, es posible ampliar el término para incluir otros factores del contexto de ejecución de programas, como ser la arquitectura de *hardware*, disponibilidad de periféricos, etc. Ejemplos de plataforma son: Microsoft Windows (diferentes versiones), GNU/Linux (diferentes versiones), IBM AIX, etc.

- **Servicio:** Un servicio, entendido en el contexto del desarrollo de software en capas, es un conjunto de operaciones y recursos con estado interno⁷⁵, relacionados entre sí, que colaboran para obtener un determinado fin. Un servicio es utilizado por un cliente a través de su interfaz y cuenta con al menos una implementación concreta de la funcionalidad que de él se espera. En estos términos, un servicio es una abstracción de un conjunto de operaciones primitivas que una capa provee a una capa superior o al programador[Tan03]. Ejemplos de servicios son: Concurrencia (i.e. *Threads*, *Mutex*, etc), Comunicaciones (i.e. *sockets*), Comunicación entre procesos (i.e. colas de mensajes, memoria compartida, etc). En la sección 2.2 se describen algunos servicios que podría proveer un CPSM.

⁷⁵Entiéndase por recursos tanto objetos explícitos de una clase determinada, que son manipulados invocando sus métodos públicos, cuanto objetos implícitos, accedidos mediante un *handle* (e.g. un *file descriptor*)

Referencias

- [Ame03] American National Standards Institute. Programming Languages C++. 2003.
- [BA90] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall, 1990.
- [Bar05] Jonathan Bartlett. The art of metaprogramming, Part 1: Introduction to metaprogramming. 2005.
- [Bin08] Andrew Binstock. Choosing Between Synchronization Primitives. *Intel Software Network*, 2008.
- [Clu05] Don Clugston. Member Function Pointers and the Fastest Possible C++ Delegates. 2005.
- [DAR08] Beman Dawes, David Abrahams, and Rene Rivera. Boost C++ Libraries. *www.boost.org*, 2008.
- [Daw08a] Beman Dawes. *Boost Filesystem Library*. Boost C++ Libraries. *www.boost.org*, 2008.
- [Daw08b] Beman Dawes. Boost Library Requirements and Guidelines. *www.boost.org*, 2008.
- [Dig96] Digital Equipment Corporation. Programming with ONC RPC. *www.hp.com*, 1996.
- [Dre05] Ulrich Drepper. Dictatorship of the Minorities. *udrepper.livejournal.com*, 2005.
- [Dre08] Ulrich Drepper. Futexes Are Tricky. *Red Hat Inc.*, 2008.
- [ea08] Rene Rivera et al. Who's Using Boost? *www.boost.org*, 2008.
- [Fog07] Agner Fog. Calling conventions for different C++ compilers and operating systems. *Copenhagen University College of Engineering*, 2007.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [His00] Larry Hiscock. About NSPR. *Mozilla Developer Center*, 2000.
- [His04] Larry Hiscock. Using IO Timeout And Interrupt On NT. *Mozilla Developer Center*, 2004.
- [IEE04] IEEE. Std 1003.1. *The Open Group Base Specifications Issue 6*, 2004.
- [Iqb] Tabrez Iqbal. Boost Filesystem Library: Writing Portable C++ Programs to Access The Filesystem. *beans.seartipy.com*.
- [JTC03] JTC1/SC22/WG21 - The C++ Standards Committee. Library Technical Report. 2003.
- [Kem06] William E. Kempf. *Boost.Thread*. Boost C++ Libraries. *www.boost.org*, 2006.
- [Koh08] Christopher M. Kohlhoff. *Boost.Asio*. Boost C++ Libraries. *www.boost.org*, 2008.

- [Lana] Sam Lantinga. SDL - Frequently Asked Questions: General. *www.libsdl.org*.
- [Lanb] Sam Lantinga. SDL - Introduction. *www.libsdl.org*.
- [Lan05] Sam Lantinga. Simple DirectMedia Layer. *www.libsdl.org*, 2005.
- [LB96] Bil Lewis and Daniel J. Berg. *PThreads Primer: A Guide to Multithreaded Programming*. Prentice-Hall, 1996.
- [Mar06] Marshall Cline. C++ FAQ Lite. 2006.
- [MDM⁺02] Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.
- [Mey97] Bertrand Meyer. *Object-oriented software construction*. Prentice-Hall, 2nd edition, 1997.
- [Mic06] Microsoft Corporation. Inside I/O Completion Ports. *Microsoft TechNet*, 2006.
- [Mic08a] Microsoft Corporation. Mutex Objects. *msdn2.microsoft.com*, 2008.
- [Mic08b] Microsoft Corporation. SignalObjectAndWait Function. *msdn2.microsoft.com*, 2008.
- [Mic08c] Microsoft Corporation. _spawnv. *msdn2.microsoft.com*, 2008.
- [Mic08d] Microsoft Corporation. Using Event Objects. *msdn.microsoft.com*, 2008.
- [Mic08e] Microsoft Corporation. Windows Sockets Error Codes. *msdn2.microsoft.com*, 2008.
- [Mic08f] Microsoft Corporation. WSASStartup Function. *msdn2.microsoft.com*, 2008.
- [Min04] MinGW - Minimalist GNU for Windows. MinGW - Frequently Asked Questions: How can I build a cross compiler? *http://www.mingw.org*, 2004.
- [Min07] Minimalist GNU for Windows. Windows sockets versus Unix sockets. *www.mingw.org*, 2007.
- [Moz] Mozilla Foundation. Netscape Portable Runtime. *www.mozilla.org*.
- [Moz98] Mozilla Foundation. NSPR: Build System and Supported Platforms. *www.mozilla.org*, 1998.
- [Moz06a] Mozilla Foundation. NSPR 4.6.4 Release. *www.mozilla.org*, 2006.
- [Moz06b] Mozilla Foundation. NSPR Contributor's Guide. *www.mozilla.org*, 2006.
- [Moz07a] Mozilla Foundation. XPCOM API Reference - nsIFile. *Mozilla Developer Center*, 2007.
- [Moz07b] Mozilla Foundation. XPIDL. *developer.mozilla.org*, 2007.
- [Moz08a] Mozilla Foundation. NSPR Reference. *www.mozilla.org*, 2008.
- [Moz08b] Mozilla Foundation. Windows Build Prerequisites. *developer.mozilla.org*, 2008.

- [Obj] Object Management Group. OMG IDL: Details. *www.omg.org*.
- [Obj04] Object Management Group. *Common Object Request Broker Architecture: Core Specification*, chapter 3. *www.omg.org*, 3.0.3 edition, 2004.
- [Par01a] Rick Parrish. XPCOM Part 1: An introduction to XPCOM - Component architecture by Mozilla. *IBM developerWorks*, 2001.
- [Par01b] Rick Parrish. XPCOM Part 5: Implementation. *IBM developerWorks*, 2001.
- [Pet04] Johan Petersson. When standards collide: the problem with dlsym. 2004.
- [RMSS06] Roland McGrath Richard M. Stallman and Paul D. Smith. GNU Make: A Program for Directing Recompilation. *http://www.gnu.org*, 2006.
- [Sal] Jon Salz. NMSTL for C++ API documentation.
- [Sch92] Douglas C. Schmidt. C++ Wrappers for Efficient, Portable, and Flexible Network Programming. *C++ Report*, 1992.
- [Sch93] Douglas C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. *Sun User Group*, 1993.
- [Sch97] Douglas C. Schmidt. Acceptor-Connector: An Object Creational Pattern for Connecting and Initializing Communication Services. *Pattern Languages of Program Design 3*, 1997.
- [Sch99a] Douglas C. Schmidt. Strategized Locking, Thread-safe Interface, and Scoped Locking: Patterns and Idioms for Simplifying Multi-threaded C++ Components. Technical report, 1999.
- [Sch99b] Douglas C. Schmidt. Wrapper Facade: A Structural Pattern for Encapsulating Functions within Classes. *C++ Report*, 1999.
- [Sch06] Douglas C. Schmidt. Overview of ACE. 2006.
- [Ses00] Peter Sestoft. *Java Precisely*. IT University of Copenhagen, Denmark, 2000.
- [Sla01] Slashdot. Sam Lantinga Slings Some Answers. 2001.
- [SP] Douglas C. Schmidt and Irfan Pyarali. Strategies for Implementing POSIX Condition Variables on Win32.
- [Spe87] Henry Spencer. Ten Commandments For C Programmers. 1987.
- [Spe90] Henry Spencer. The Ten Commandments for C Programmers (Annotated Edition). 1990.
- [SRZea07] Julian Smart, Robert Roebing, Vadim Zeitlin, and Robin Dunn et al. wxWidgets 2.8.7: A portable C++ and Python GUI toolkit. *www.wxwidgets.org*, 2007.
- [SRZea08a] Julian Smart, Robert Roebing, Vadim Zeitlin, and Robin Dunn et al. About the wxWidgets Project. *www.wxwidgets.org*, 2008.

- [SRZea08b] Julian Smart, Robert Roebling, Vadim Zeitlin, and Robin Dunn et al. wxWidgets Features. *www.wxwidgets.org*, 2008.
- [SRZea08c] Julian Smart, Robert Roebling, Vadim Zeitlin, and Robin Dunn et al. wxWidgets Users. *www.wxwidgets.org*, 2008.
- [SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley and Sons, 2000.
- [Sta06] William Stallings. *Cryptography and Network Security*. Prentice Hall, 4th edition, 2006.
- [Str01] Bjarne Stroustrup. *Programming with Exceptions*. 2001.
- [Sun08] Sun Microsystems. The Java HotSpot Performance Engine Architecture. *Sun Developer Network*, 2008.
- [Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.
- [Tan02] Andrew S. Tanenbaum. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.
- [Tan03] Andrew S. Tanenbaum. *Redes de computadoras*. Prentice Hall, 4th edition, 2003.
- [The06] The FreeBSD Documentation Project. *FreeBSD Developers' Handbook*. 2006.
- [The08a] The C++ Standards Committee. C++ Standard Core Language Active Issues, Revision 54. 2008.
- [The08b] The C++ Standards Committee. C++ Standard Core Language Defect Reports, Revision 54. 2008.
- [TO03] Doug Turner and Ian Oeschger. Creating XPCOM Components. *Mozilla Developer Center*, 2003.
- [vH06] Dimitri van Heesch. *Doxygen Manual for version 1.5.4*. 2006.
- [VR01] Paulo Veríssimo and Luís Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publisher, 2001.