UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

UNIDAD ACADEMICA DE LOS CICLOS PROFESIONAL Y DE POSGRADO DEL COLEGIO DE CIENCIAS Y HUMANIDADES

MAESTRIA EN CIENCIAS DE LA COMPUTACION

INSTITUTO DE INVESTIGACIONES DE MATEMATICAS APLICADAS Y SISTEMAS

MECANISMOS DE INTERACCION ENTRE PROCESOS DISTRIBUIDOS CON LA METODOLOGIA ORIENTADA A OBJETOS

T E S I S

QUE PARA OBTENER EL GRADO DE

MAESTRIA EN CIENCIAS DE LA COMPUTACION

P R E S E N T A

LIUBA GINA SARMIENTO NAVA

DIRECTORA: DRA. HANNA OKTABA

MEXICO, D. F. **1995**

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

Una palabra nunca agotada :

GRACIAS

Una tesis es un requisito para cumplir un plan de estudios, es verdad; pero si alzamos la vista y el corazón, podemos observar que es un pretexto excelente para desarrollar relaciones: relaciones que tienen nombre y apellido y otras que son acercamiento a formas de trabajo y actitudes. Una tesis permite descubrir en una atmósfera de entusiasmo y consenso, de sinsabores y satisfacciones, la curiosidad investigadora, el conocimiento y sentimientos diversos.

No soy afecta a citar a la manera de una lista de escuela en la que se sitúa a cada quien con una, dos o más 'estrellitas' de acuerdo a sus cualidades, porque esto inmiscuye una cuantificación, algo por demás absurdo en las relaciones humanas porque todas ellas son diferentes; sin embargo, reconozco que existen momentos en los que debo exteriorizarse el pensamiento que uno tiene hacia los demás, tal vez éste sea uno de ellos :

Doy GRACIAS a ***DIOS*** por poner a mi alcance tantos maravillosos obsequios, de los que destaco aquel que es el más preciado que tengo y ruego conservar : ***MI FAMILIA.***

GRACIAS *HANNA*, pienso que tu forma de ser debería tomarse como materia de estudio para luego crear sistemas de producción de Hannitas y Hannitos en serie (obviamente Orientados a Objetos) porque si la tierra estuviera poblada de más gente como tú, estoy segura que las noticias serían distintas. Eres de esas personas que son de todos y no son de nadie; pero en mi corazón siempre serás mi Hanna.

A mi *JURADO* : *Fabián García Nocetti*, a *Víctor Germán Sánchez* , a *Gloria Quintanilla* y a *Lupita Iturguengoitia*, muchas GRACIAS porque venciendo obstáculos de distancia y tiempo, me hicieron sentir su apoyo.

Gracias a mis *PROFESORES* y AMIGOS, esos amigos de todos los días o de encuentros esporádicos; pero valiosos, a quienes quisiera conservar para toda la vida, a pesar de que ésta nos disperse por diferentes caminos...

Y un agradecimiento muy especial a la *SECRETARIA DE RELACIONES EXTERIORES DE MEXICO* (particularmente al Lic. Arturo Márquez), por la ayuda que me brindaron financiando parte de mis estudios. También mi gratitud para el Programa *P.A.P.I.I.D.* de la *U.N.A.M.*, por su colaboración para realizar esta tesis.

SINCERAMENTE:

TABLA DE CONTENIDO

CAPITULO 3

CAPITULO 4

CAPITULO 5

JERARQUIAS DE CLASES BASICAS PARA EL SOPORTE DE LA COMUNICACION Y SINCRONIZACION POR PASO DE MENSAJES 96

CAPITULO 6

INTRODUCCION

La relación *causa - efecto* es una verdad universal y a pesar que frecuentemente la causa es un conflicto, muchas veces el efecto se traduce en amplios beneficios, los cuales no hubieran llegado de no ser por la presión del problema. Así, las Ciencias de la Computación han evolucionado respondiendo a exigencias un tanto traumáticas, ya que la desesperación de verse limitado por agentes externos como la lentitud de las máquinas y/o dispositivos periféricos, lenguajes poco expresivos, herramientas de apoyo inexistentes, etc., han alentado la mentalidad creativa de sus profesionales para que planteeen y desarrollen soluciones.

Las soluciones (o intentos de serlo) avanzan por dos veredas : la del del *hardware* y la del *software*, aunque ambas se encuentran en el mismo camino. El *hardware* ha producido artefactos físicos eficientes (cada día más variados y rápidos), el *software* también ha obtenido avances notables, aunque debe reconocerse que de un modo más desordenado, hecho que no pasó desapercibido, provocando que a partir de los años 60 (época conocida como de la crisis del software) empezara a enfocarse a nuevas formas de concepción e implantación de sistemas.

Existen propuestas completamente innovadoras, que actúan como verdades reveladoras de nuevas perspectivas, otras tratan de combinar las ventajas comprobadas de estas verdades. El trabajo de esta tesis : ***Mecanismos de Interacción entre Procesos Distribuidos con la Metodología Orientada a Objetos***, hace un planteamiento gravitando alrededor de los modelos de programación Concurrente - Distribuida y del paradigma Orientado a Objetos. Su objetivo es crear patrones de interacción entre procesos de programas Distribuidos, mediante mecanismos lógicos de diferentes niveles de complejidad, se trata de estandarizar dichos patrones, haciéndolos generales, independientes de aplicaciones particulares y a la vez flexibles a la adecuación a problemas específicos.

La explicación detallada de los mecanismos modelados en el trabajo de tesis se encuentra en los **capítulos 5 y 6**, en el primero se abordan los mecanismos básicos que sirven de soporte para los mecanismos considerados en el segundo (todos ellos expresados en seudocódigo bastante cercano al lenguaje C++Concurrente del M. en C. José Oscar Olmedo, a quien agradecemos nos facilitara una copia de su prototipo, para fines de prueba de los mecanismos elaborados). Para llegar a estos capítulos con el bagage de conocimientos necesarios y sobre todo uniformidad de términos, dado que de acuerdo a las fuentes consultadas, existe disparidad en su definición, se incluyen cuatro capítulos. El **capítulo 1** revisa los principales conceptos de la Metodología Orientada a Objetos; el **capítulo 2** registra los puntos esenciales de los Sistemas Concurrentes y Distribuidos; el **capítulo 3** expone las ventajas y problemas de combinar Concurrencia, Distribución y Orientación a Objetos; finalmente, el **capítulo 4** introduce el trabajo realizado y lo sitúa en el contexto deseado, haciendo las veces de nexo entre el marco teórico de los capítulos previos y el desarrollo de los capítulos 5 y 6.

CAPITULO 1

PENETRANDO EN EL MUNDO DE LA ORIENTACION A OBJETOS

¿Qué significa ser o estar Orientado a Objetos? ¿Quién o qué es Orientado a Objetos?, ¿Cuál es el origen del término?, ¿Hasta dónde llega? ... Son muchas las dudas y quizás más aun las respuestas; son tantas, que conforman un oceano al cual hay que zambullirse para rescatar los conceptos más importantes, aquellos que realmente dan una idea justa de lo que es la Orientación a Objetos. Este capítulo conduce al lector por un breve recorrido a través del mundo de los objetos, en un intento de dar de él, una noción consistente.

1. Antecedentes históricos

Haciendo una analogía con la evolución de la arquitectura (hardware) de computadoras, podemos ver que desde el inicio, los ingenieros electrónicos aprovecharon los conocimientos existentes. Cada generación de profesionales sólo mejoraba, optimizaba, pulía el trabajo de sus antecesores. Hoy en día, los circuitos integrados pueden armarse en cualquier diseño pues sus entradas y salidas son perfectamente conocidas.

Los ingenieros de software, por contraste, han evolucionado de la programación por cableado al lenguaje ensamblador, y luego a la programación estructurada; pero los desarrollos aplicativos, siguen siendo en general realizados de la misma manera : desde cero. No existe un esquema de aprovechamiento de lo desarrollado previamente [Archundia 92].

Los investigadores, conociendo este problema, comienzan a enfocar esfuerzos a esta problemática a principios de los setentas. La teoría de la ***Programación Orientada a Objetos*** **(POO)** se desarrolla como respuesta coherente.

PARC Xerox, en 1972, como parte de un ambicioso proyecto de software y harware (*Dynabook*, a cargo de Alan Kay), presenta un lenguaje de programación al que llama ***Smalltalk*** . Diseñado con características especiales en mente, introduce conceptos **Orientados a Objetos**, como se ha dado en llamarles. Sin embargo, para ser justos, fue ***Simula*** (de Dahl y Nygaard), en los años sesentas, quien por primera vez incluye el concepto de **"objeto"**. Aun así, la historia de *Smalltalk* se toma como la historia de la POO.

Actualmente existen varios lenguajes que incorporan el modelo de *Objetos* : versiones de *Smalltalk, Eiffel, ABCL, C++, Object Pascal, Emerald,* etc., cada uno con propias tendencias, alcances y limitaciones; pero todos con la promesa de permitir un estilo de programación que reduzca los problemas de concepción e implantación de grandes sistemas de software.

2. ¿Qué conceptos implica la Orientación a Objetos ?

Se omitió intencionalemente en el título de este capítulo una etiqueta al término Orientación a Objetos, lo asociamos con Programación OO, Lenguajes OO, Metodos OO, Paradigma OO, etc. ***¿A qué se refiere exactamente?***. La literatura computacional juega bastante con las palabras; muchos libros, revistas o productos enumeran una serie de características OO y las presentan bajo diferentes rotulos, ¿Acaso alguno miente?. No, tal vez normalmente exageren; pero éste no es el lugar ni el momento para entablar un juicio ético. Aquí, lo que se desea es establecer el origen de la confusión y una solución. Pensamos que todos tienen su verdad porque ciertamente los conceptos OO han invadido bastantes áreas, siendo entonces válido que cada una de ellas se adueñe del término. Las caracterísiticas son las mismas; pero en un orden de prioridad diferente y con matices convenientes según el sector.

El presente trabajo, a lo mejor cayendo en lo mismo que critica, realiza una clasificación más, esta vez agrupando las características según tres puntos de vista, el primero y el segundo se identifican con lo que Warren Greiff, en su serie de artículos [Greiff 93:a,b y 94], denomina características OO como paradigma y características para la Ing. de Software[1], respectivamente; mientras que el tercero se adapta a lo que Booch [Booch 94] llama elementos menores :

- Características determinantes,
- características consecuentes y/o auxiliares y
- características adicionales del modelo OO.

Las características determinantes se mueven en un marco conceptual y las consecuentes y auxiliares, en un ambiente más pragmático; individual y conjuntamente forman la metodología OO[2]. Las características adicionales no son esenciales; pero sí muy útiles cuando se integran a un paradigma dado, en este caso al OO.

2.1. Características determinantes del modelo OO

Las **características determinantes** del modelo OO son las que le dan el nivel de **paradigma**. Jose Negrete[3] dice : *Paradigma es una regla cualitativa que agrupa y/o integra a un conjunto de reglas cuantitativas*. La definición que utilizaremos, más intuitiva y filosófica, dice : *Paradigma es nuestra visión del mundo*. Nos recuerda el dicho tan conocido y verdadero "el mundo es del color del cristal con que se mira". En nuestro mundo de la computación, con el cristal de este nuevo paradigma, *un sistema está poblado de entes con estados internos comunicándose unos con otros mediante mensajes, en pos de un fin común* [Greiff 93:b]. La organización gira en torno de los seres o más bien de sus capacidades de acción y no a las acciones por sí solas.

[1]Greiff introduce estas características como la metodología OO.
[2]En este trabajo se discrepa con Greiff respecto al término metodología OO. Para nosotros esta palabra implica tanto las características OO desde el punto de vista de paradigma, como las de Ing. de Software; las primeras sientan las bases para que las segundas existan o se expresen de cierta manera; ambas, marcan una metodología para resolver un problema desde su planteamiento hasta su implantación. Greiff presenta ambas características como dos caras de la Programación OO.
[3]Investigador del área de Biomédica de la UNAM. Miembro Fundador de la Sociedad Mexicana de IA.

Los elementos que identifican a la Orientación a Objetos como un paradigma son :

- **Clases y objetos :**

Una clase es la implementación de un TDA [Oktaba y Quintanilla 93]. Tiene las misma ventajas y flexibilidad adicional. Representa al ente que modela a través de atributos y métodos que determinan su comportamiento. Consta de una parte de exhibición (interfaz), que es el escaparate de los servicios que ofrece a otras clases y una parte de definición de todos sus métodos (los declarados en la interfaz son un subconjunto del conjunto total).

Un objeto es la instancia de una clase, contiene valores para sus atributos y envía mensajes a otros objetos para interactuar cooperativamente (relación de *uso*). Es el 'ser vivo', creado con el esqueleto de la clase que lo describe.

- **Herencia :**

Muchos de los seres del mundo real están vinculados por relaciones de parentesco, igualmente en la abstracción de dicho mundo, las *clases* se relacionan por herencia (relación *es un*).

Gracias al mecanismo de herencia, una clase del sistema recibe atributos y servicios de otra u otras, convirtiéndose en su descendiente. El *hijo reutiliza la tecnología aplicada en el desarrollo de la clase padre y expande o especializa su estado y funcionalidad* al añadir nuevas partes. A su vez, esta clase puede ser heredada por otras, creando una jerarquías de clases que eventualmente formarán una biblioteca de diseños o en su defecto de código reutilizable. Al liberar una clase a esa biblioteca, se la considera totalmente probada y con su comportamiento íntegramente descrito.

- **Mensajes :**

El envío de un mensaje a un objeto para que responda con alguna acción, se traduce a la invocación de un método o mejor aún al llamado de alguna función de ese objeto. Sin embargo, en un mundo de seres en comunicación resulta difícil acomodar la idea de función; es más natural pensar en término de mensajes. Parece una diferencia sutil; pero es considerable si se busca cambiar el modo de enfocar los problemas.

2.2. Características consecuentes y/o auxiliares del modelo OO

Las características consecuentes y/o auxiliares son derivaciones del paradigma OO que enriquecen el espectro de opciones para el desarrollo de software de calidad. Son parte integral del modelo OO; pero no imprescindibles, al punto que su ausencia pudiese mutilar al programador de su capacidad de pensar un sistema en *Objetos*. Como son características que facilitan las etapas del ciclo de vida de un producto de software, se piensa en ellas como características enfocadas a la Ingeniería de Software.

Estas características son[4] :

- **Abstracción** : extrae las características esenciales de algún objeto (relativas a la perspectiva del observador) y las plasma en la estructura descriptiva de clase.
- **Encapsulación** : oculta los detalles de un objeto que no contribuyen a sus características esenciales, colocándolos fuera de la interfaz visible del Objeto.
- **Modularidad** : propiedad de descomponer un sistema en módulos debilmente acoplados y altamente cohesionados. Los módulos no se basan en procedimientos sino en Objetos.
- **Jerarquía** : estructura en la que sus elementos (clases) guardan un orden, rango o categoría, de acuerdo a ciertas referencias de clasificación. Es una consecuencia directa de la propiedad de herencia.
- **Control de tipos** : evita mezclar diferentes abstracciones.
- **Polimorfismo** : permite que una operación tenga diferentes conductas en objetos distintos, o dicho de otro modo, distintos objetos reaccionen de manera diferente al mismo mensaje.
- **Sobrecarga** : hace posible definir varias funciones con el mismo nombre.
- **Liga dinámica** : otorga la facilidad de especializar una función en tiempo de ejecución, de acuerdo al tipo del Objeto sobre el que se manda el mensaje.
- **Funciones diferidas** : son funciones declaradas en una clase y definidas en su(s) descendiente(s). Sirven para especificar servicios que deben ser proporcionados por todos los herederos.
- **Genericidad** : faculta crear clases parametrizadas por uno o mas tipos de datos (otras clases).

[4]Varias definiciones se tomaron del artículo [Quintanilla 93].

- **Administración automática de memoria** : liberación del espacio ocupado por Objetos que ya no son referenciados por ningún otro. Exentar de esta responsabilidad al programador hace más confiable al sistema; pero puede acarrear ineficiencia y mayor margen de error.

2.3. Características adicionales del modelo OO

La concurrencia y la **persistencia** son propiedades que abren nuevas perspectivas y poder para la solución de problemas. El modelo de objetos no es ajeno a sus posibles ventajas y como otros paradigmas de computación, también intenta incorporarlas sin contradecir sus postulados, no como características primordiales del paradigma; pero sí como características adicionales. Por otra parte, debe recordarse que ambas están presentes en el mundo real y si el modelo OO presume de mapearlo correctamente, resulta lógico que las tome en cuenta. A continuación se cita la función de cada una de ellas :

- **Concurrencia** : permite a diferentes Objetos actuar al mismo tiempo[3].

- **Persistencia** : salva las clases y el estado de los Objetos en el tiempo y el espacio. Los valores no se pierden a pesar de concluir la ejecución del programa.

3. Un lenguaje de programación rumbo a la Orientación a Objetos

Bobrow and Stefick definen *el estilo de programación como una forma de organizar los programas sobre las bases de algún modelo conceptual de programación y un lenguaje apropiado para hacer programas escritos con un estilo claro* [Bobrow y Stefick 86]. Hablar de un lenguaje apropiado es un tanto subjetivo porque muchas veces depende de la capacidad y costumbre del programador; sin embargo, desde el punto de vista del problema, éste puede ser atacado de mejor manera atendiendo a su naturaleza, para así elegir un lenguaje que tenga características en ese sentido, por ejemplo un lenguaje orientado a procedimientos (algorítmico), orientado a objetos (con clases y objetos), lógico (con metas y cálculo de predicados), orientado a reglas (if-then), orientados a restricciones (con relaciones invariantes), funcional (con funciones), etc.

Dividiendo a los lenguajes dentro de los paradigmas no estructurados, estructurados y OO, se obtiene un mapeo del mundo real que va de menos a más. No por sí mismos, sino porque responden a un determinado paradigma; de hecho, los lenguajes son su tarjeta de presentación. Los lenguajes no son el paradigma completo; pero sí lo constituyen, son los edecanes más socorridos para difundirlo. Mucha gente llega a ser productivo utilizando un lenguaje, desconociendo otra cara del paradigma.

[3]Este aspecto se retoma a lo largo de todo el trabajo.

Es importante trabajar con un lenguaje apropiado. La consistencia de ideas y hechos es primordial, asegura un mejor resultado -en la propia vida y más mundanamente- en la concepción de un sistema de cómputo. Se publicitan las bondades de la metodología OO, bueno, es ideal explotarla en todos sus ámbitos : pensar un producto en términos OO y concretarlo con un lenguaje OO.

El modelo OO está en boga, consecuentemente en la actualidad muchos lenguajes y sistemas dicen serlo. Como lo presenta la Mercadotecnia, acercarse al paradigma OO parece sinónimo de alcanzar la "tierra prometida"; o sinónimo de pertenecer a un linaje real dentro del entorno computacional, pues bien según los expertos :

> ¿Cúales son las pruebas que debe pasar un ***lenguaje*** para ser considerado ***de sangre azul***?

Sintéticamente, un lenguaje para ser llamado OO debe satisfacer los siguientes requerimientos :

Según Cardelli y Wegner [Cardelli y Wegner 85][6] :

- Soporte de objetos que son abstracción de datos con una interfaz de nombres de operación y estados locales escondidos.
- Los Objetos tienen un tipo asociado (clase).
- Los tipos (clases) pueden heredar atributos de supertipos (superclases).

Meyer es más extenso en sus condiciones involucrando algunas caracteríaticas consecuentes y/o auxiliares [Meyer 88] :

- Estructura modular basada en Objetos.
- Abstracción de datos (los objetos son descritos como implementaciones de TDA's).
- Manejo de memoria automático.
- Clases (cada tipo no simple es un módulo y cada módulo de alto nivel es un tipo, para eliminar a los tipos simples y a los procedimientos).
- Herencia.
- Polimorfismo y ligado dinámico.
- Herencia múltiple y repetida.

En el sentido estricto de Meyer y Wegner[7], los lenguajes que soportan herencia son llamados *Orientados a Objetos*, los que carecen de ella, *Basados en Objetos*.

[6]Citado en el capítulo 2 de [Booch 94].
[7]En [Wegner 87] y [Wegner 92].

4. La Orientación a Objetos va más allá de un lenguaje de programación

Dada la trascendencia de la Orientación a Objetos, no únicamente los lenguajes adoptaron sus postulados, también otros sectores.

Dentro de la Ingeniería de Software, que es útil para abordar el desarrollo del software de cualquier área computacional, el modelo OO es importante en las etapas de análisis y diseño [Oktaba 93:a,b]. En el análisis OO se examinan los requerimientos dentro del vocabulario del dominio del problema, desde una perspectiva de clases y objetos. En el diseño OO se hace énfasis en el proceso de descomposición OO y la notación empleada para expresar diferentes modelos del diseño lógico (estructura de clases y objetos) y físico (arquitectura de módulos y procesos) de un sistema. Las etapas posteriores (implantación, prueba y mantenimiento) reducen sus costos como consecuencia de un buen análisis y diseño y del empleo de lenguajes y herramientas OO.

El estudio del modelo OO en cuanto a la compatibilidad de conceptos, ventajas y contradicciones, prácticas y teóricas, en ámbitos tan variados como la creación de interfaces gráficas, construcción de herramientas de 4ta. generación, Bases de Datos, Inteligencia Artificial, Sistemas Multimedia, Concurrentes, Distribuidos, etc., ya ha dado resultados tangibles, aunque todavía es campo fértil para la investigación.

CAPITULO 2

CONCURRENCIA Y DISTRIBUCION

Es un reto general obtener más con menos. Los sistemas computacionales no son la excepción, buscan mayor expresividad, rapidez, efectividad y otros, con menor desperdicio de dispositivos, esfuerzo, tiempo, etc.

Dejar el trabajo a una sola entidad parece no ser suficiente para las exigencias actuales; por tanto, la investigación propone dividir la faena entre varios responsables. Lo que se alza como una solución acarrea nuevos problemas : son necesarias reglas que rijan la convivencia de esta nueva sociedad. El estudio de la interación entre sus miembros origina los sistemas concurrentes y sus derivaciones.

Este capítulo introduce los principales conceptos de los sistemas concurrentes, seudoparalelos, paralelos y distribuidos, los elementos necesarios para soportarlos, sus puntos en común y diferencias.

1. Introducción

Un sistema de computación abarca componentes que trabajan a distintas velocidades; cuando están dedicadas a un sólo *programa en ejecución* (un *proceso*), lamentablemente son frecuentes los tiempos muertos del procesador, por ejemplo durante la espera de datos introducidos a partir de un periférico como el teclado. Para aminorar este despercicio de recursos surge la idea de concurrencia, primero con la ***multiprogramación*** y luego con la ***programación multitarea***.

Multiprogramación es la ejecución de varios programas independientes en un procesador, en un mismo período de tiempo, alternando los lapsos de procesamiento y suspención entre los distintos programas. Extendiendo la idea : si el procesador puede ser compartido por varios programas, ¿Por qué no compartirlo entre diversas partes de un mismo programa?. La respuesta la otorga la ***programación multitarea***, en ella un problema se mapea a un programa desglozado en fragmentos (tareas) que pueden ser vistos como nuevos programas (procesos), esta vez no independientes, sino cooperantes en pos de una solución común. Para que interactuén apropiadamente deben ajustarse a algún modelo establecido por la programación concurrente.

2. Programación concurrente

La programación concurrente, según algunos autores (por ejemplo [Bal, *et. al.* 89]), es sinónimo de ***seudoparalelismo***, es decir ejecución de varios procesos en igual intervalo de tiempo mas no al mismo tiempo -***paralelismo***. Para nosostros, programación concurrente es más amplia, es un modelo conceptual[1] de descomposición de un problema en términos de entidades interactuando en paralelismo abstracto, después traducible a paralelismo real (varios procesadores o computadoras), seudoparalelismo (paralelismo simulado mediante un único procesador) o una combinación de ambos.

2.1. Definición

> ***Un programa concurrente*** *especifica dos o más procesos que cooperan en la ejecución de algún problema. Cada proceso es un programa secuencial que ejecuta una secuencia de instrucciones y se comunica con los demás para cooperar. La comunicación se realiza mediante variables compartidas o por el paso de mensajes* [Andrews 91].[2]

La comunicación entre los procesos no se realiza arbitrariamente, responde a esquemas de sincronización preestablecidos. La ***comunicación y la sincronización*** son aspectos íntimamente ligados en lo que denominaremos ***mecanismos de interacción***.

[1] Es un paradigma, en el sentido explicado en el capítulo anterior.

[2] En este contexto, la palabra proceso se usa para los programas secuenciales y el término programa para el conjunto de procesos.

2.2. Mecanismos de comunicación y sincronización con memoria compartida

La comunicación con memoria compartida se realiza a través de variables comunes a los procesos, uno o más procesos escriben en ellas mientras que otro u otros las leen. Estos mecanismos evolucionaron de un menor a un mayor grado de abstracción en dirección de una liberación del programador de detalles engorrosos. El objetivo de algunos mecanismos se reduce a garantizar la *exlusión mutua*, otros más potentes, permiten la *sincronización condicional*. Como el presente trabajo no se basa en este tipo de mecanismos, sólo son mencionados[3] :

Algortimos de 'busy waiting'.
Semáforos.
Regiones críticas.
Monitores.

2.3. Mecanismos de comunicación y sincronización por paso de mensajes

Cuando se utiliza el paso de mensajes para coordinar la actividad concurrente de varios procesos, unos procesos envían información y otros la reciben. Unos son emisores y otros, receptores. La comunicación de un mensaje se da únicamente si los procesos están sincronizados acordes a algún modelo (síncrono o asíncrono)[4].

2.4. Necesidades de expresión en un lenguaje Concurrente

Todos los lenguajes incorporan construcciones especiales que los caracterízan, los lenguajes concurrentes, no importa el modelo y el estilo que sigan, deben incluir alguna forma de introducir la concurrencia (ésta puede en un momento dado comprometer la elección de algún mecanismo de interacción en especial) y el no determinismo.

2.4.1. Expresión de la concurrencia

- **Procesos** : en muchos lenguajes la concurrencia está basada en la noción de proceso. El concepto de ***proceso*** incialmente ligado a los lenguajes procedurales concurrentes, alcanza una connotación general como sinónimo de ***unidad de concurrencia***. En este contexto, con visos particulares, es citado bajo diferentes paradigmas de programación que desean incluir propiedades de concurrencia.

[3] Para una consulta detallada se pueden revisar [Ari 90][Oktaba 88,c,d] [Andrews 91].
[4] La interacción vía mensajes se amplía al describir los Sistemas Distribuidos (posteriormente en este capítulo) y en el resto del trabajo, porque se centra en ella.

Según un concenso más o menos general, el proceso es visto como un procesador lógico que ejecuta código secuencialmente (cuando tiene un solo hilo de control). Es la contraparte dinámica de un programa.

Un proceso tiene un espacio de nombres que delimita las entidades que le pertenecen, así como los permisos para acceso y uso de recursos del sistema. Está protegido por fronteras que lo apartan de los otros procesos, posee un estado de ejecución (*ready*, *suspended*, etc.) y posiblemente una prioridad. También tiene una parte dinámica conocida como *thread* (hilo de control), el cual deja la huella o *'trace'* representando el camino de ejecución que está siendo atravesado.Como se sugiere en [Rashid 88], la abstracción de proceso puede dividirse en un ambiente que es la unidad básica de asignación de recursos y un *thread* que es la unidad básica de utilización de *CPU*, correspondiente a la noción intuitiva de actividad [Pedregal 89].

Cuando se permiten varios *threads* en el mismo ambiente, compartiendo el mismo espacio de direccionamiento, se tiene concurrencia interna al proceso, Estos hilos, llamados también procesos peso ligero[5], son como 'mini-procesos', cada uno con su propio contador de programa y stack[6]. Tanto un proceso como los 'mini-procesos' pueden ser creados implícita o explícitamente.

Haciendo analogías, si el *thread* es la abstracción del procesador, *multithreating* en un proceso es como tener un multiprocesador con memoria central compartida.

- **Objetos** : los lenguajes Orientados a Objetos secuenciales se basan en un modelo de objetos pasivos. Un objeto se activa únicamente cuando reciben el mensaje de otro. En cualquier momento sólo puede haber un solo objeto activo a la vez. Surgen propuestas que extienden el modelo en varias direcciones[7].

- **Enunciados paralelos** : Otra forma de expresar el paralelismo es agrupando instrucciones (cada una por sí conforma un proceso), con ciertos constructores, por ejemplo. :

  - **PAR**
    secuencia1
    secuencia2 Ambas secuencias se desarrollan en paralelo.

  - **PAR** i = 0 **FOR** n
    A[i] := A[i] + 1 Todas las iteraciones del ciclo se hacen en paralelo.

[5] *Lightweight processes* comparten el espacio de direccionamiento, los que no lo hacen son llamados *heavyweight processes* [Andrews 91].
[6] Se recomienda consultar [Tanembaun 92] para mayores detalles.
[7] Este es tema del próximo capítulo.

- **Paralelismo funcional** : retoma la idea matemática de las funciones, aquella en la que una función depende solamente de sus valores de entrada y carece de efectos colaterales. Por ejemplo, dada :

  funA (*funB*(4,9), *funC*(7)) es irrelevante el orden en que se calculen las funciones enviadas como argumento a *funA*, en consecuencia es posible hacerlo en paralelo.

- **Paralelismo AND/OR** : Por ejemplo :

  A :- B, C, D.
  A :- E, F.

  Es decir, si los subteoremas B, C y D son verdaderos, entonces A es verdadero, o bien, A es verdadero si E y F son verdaderos. Resultan evidentes dos oportunidades para introducir paralelismo en la programación lógica :

  - Las dos cláusulas para A pueden ejecutarse en paralelo hasta que una de ellas se satisfaga o ambas fallen (paralelismo OR).

  - Para cada una de las cláusulas, los subteoremas trabajen en paralelo, hasta que todos se cumplan o alguno falle (paralelismo AND).

2.4.2. Expresión del no determinismo

La interacción entre los procesos no necesariamente sigue un orden determinado. Existen puntos que conglomeran a varios procesos en espera de algún mensaje o servicio, si sólo se puede atender a uno de ellos, se debe elegir a un agraciado. Esta elección puede responder a prioridades, depender de condiciones en tiempo de ejecución, ser completamente no determinista (aleatoria) o una conjunción de estas posibilidades.

El no determinismo se puede controlar con estructuras de selección del estilo del comando custodiado de Dijkstra [Dijkstra 68] y sus derivaciones hechas por Brinch Hansen [Hansen 78] y Hoare [Hoare 85].

3. Sistemas distribuidos

Autores como Ari [Ari 90] y Andrews [Andrews 91] (este trabajo adopta su tendencia), centran los requisitos de computación distribuida en la forma de comunicación y sincronización de los procesos, antes que en la ubicación física de los mismos, que viene a ser como una consecuencia del modelo de interacción. Afirman :

3.1. Definición

> ***Programas distribuidos** son **programas concurrentes** en los cuales los procesos se **comunican por el paso de mensajes**. Pueden ser fácilmente implementados en procesadores físicamente distribuidos, de ahí el nombre dado a este campo; sin embargo, también pueden ejecutarse sobre multiprocesadores con memoria compartida o aun sobre un monoprocesador.*

Bal, Steiner y Tanenbaum [Bal, *et. al.* 89] van un poco más allá, clasificando los sistemas de computación distinguen dos niveles de distribución :

- **Nivel físico** : son distribuidos si están montados en ***hardware sin memoria compartida*** (redes de computadoras); no distribuidos en otro caso (en sistemas multiprocesadores o uniprocesadores)

- **Nivel lógico** : son distribuidos cuando el ***software*** consiste de multiples *procesos* que se *comunican por el paso explícito de mensajes*; no distribuidos si la comunicación se realiza a través de datos compartidos.

De acuerdo a estos dos niveles son viables las cuatro combinaciones : sistemas corriendo en hardware físicamente distribuido/nodistribuido con software lógicamente distribuido/no distribuido.

La ***concurrencia es algo inherente a los sistemas distribuidos***, tanto bajo la forma de seudoparalelismo como paralelismo, ya que ambos se pueden combinar. Por ejemplo, si se cuentan con seis procesos y sólo tres procesadores, seguramente algunos procesos estarán físicamente distribuidos y en paralelismo real, otros deberán correr en un mismo procesador con paralelismo simulado. De esta manera, por razones de eficiencia, ***se pueden utilizar conjuntamente mecanismos de interacción de sistemas distribuidos con aquellos de memoria compartida***. Además de las técnicas centralizadas como un recurso para lograr eficiencia, el propio modelo de programación con datos compartidos no ha sido desechado de la programación distribuida; basta observar que existen propuestas de crear lenguajes distribuidos que presentan al usuario una imagen de un sistema con espacios de direccionamiento lógicamente compartido.

3.2. El concepto de granularidad

La noción de **grano** refleja el tamaño y estructura de las unidades de programa, se refiere al tamaño de las tareas (en programación concurrente) y al monto de procesamiento entre dos comunicaciones (en el entorno de programación distribuida)[8]. Es importante establecer su dimensión porque se puede traducir en beneficios o inconveniencias : granularidad fina indica tareas concurrentes pequeñas, comunicaciones muy frecuentes, expresión de detalles de concurrencia más finos y gran explotación del paralelismo; sin embargo, llevada a la exageración puede causar *overhead* sobre todo en sistemas distribuidos debilmente acoplados[9], para los que se aconseja granularidad aspera.

3.3. Características de los Sistemas Distribuidos

El soporte para la programación distribuida puede estar a nivel del Sistema Operativo o de los lenguajes. Los lenguajes representan un nivel mayor de abstracción, son más flexibles y otorgan legibilidad y portabilidad mejoradas, también verificaciones más complejas. Las características que deben cubrir son :

3.4. Mapeo de entidades lógicas (procesos) a entidades físicas (procesadores)

Se conoce por ***mapeo*** a la ubicación o *distribución de los procesos en los diferentes procesadores* de un sistema de cómputo. Esta característica es importante en sistemas físicamente distribuidos. Es una actividad trasparente para el usuario, si la realiza el compilador y el sistema en tiempo de ejecución del lenguaje, posiblemente asistidos por el sistema operativo; o programable, si está controlada por el usuario. La primera opción facilita el trabajo del programador; pero puede convertirse en una restricción para las aplicaciones porque presume un conocimiento fijo de los procesos, ignorando estrategias propias de cada problema. La segunda opción es amoldable a cada aplicación y posiblemente conduzca a una distribución más eficiente; pero significa trabajo extra para el programador.

Los procesos se pueden fijar en los procesadores a tiempo de compilación (como en StarMod [Cook 80]), a tiempo de ejecución (como en Concurrent Prolog [Shapiro 84], o a ningún tiempo porque se les permite movilidad continua (como en Emerald [Juliet, *et. al.* 88]).

[8] Wegner afirma que la granularidad de concurrencia y distribución son independientes [Wegner 92].
[9] Son los sistemas formados por computadoras ubicadas físicamente distantes. Sus antagónicos son los sistemas fuertemente acoplados.

3.6. Comunicación y sincronización por paso de mensajes

Una descripción del modelo de interacción por paso de mensajes en términos de los elementos (entes) que se ven involucrados se realiza en capítulos posteriores de este trabajo. Existen consideraciones que son independientes del modelo particular que se elija para implantar el paso de mensajes, de ellas sobresalen los siguientes aspectos :

- Si la recepción de un mensaje es explícito, es decir si el receptor ejecuta instrucciones de aceptación de determinados mensajes, quizás sujetos a ciertas condiciones; o implícito, si el código es automáticamente invocado dentro del receptor. El inicio de la comunicación es explícito mediante el envío de un mensaje o la invocación de un servicio.

- Si el paso de mensajes es direccionado en modo directo o indirecto. Es directo si se nombra específicamente a un proceso. Es indirecto si se realiza a través de un intermediario : los procesos nombran a este intermediario en lugar del emisor o el receptor.

- Si el proceso emisor del mensaje sabe a quién lo envía y recíprocamente, el receptor conoce la procedencia del mensaje, se tiene un esquema simétrico. Si sólo el proceso emisor nombra al receptor, el esquema es asimétrico.

Las formas más difundidas de interacción por paso de mensajes incluyen el paso de mensajes síncrono, el paso de mensajes asíncrono (modelados con el flujo de información en una dirección -del proceso que envía al que recibe el mensaje), el *rendezvous* y el ***Remote Procedure Call*** (modelos con comunicación bidireccional).

3.6.1. Paso de mensajes síncrono

La comunicación y la sincronización están estrechamente acopladas. Para que un mensaje pase de un proceso a otro, ambos deben acudir a una cita[10], mientras alguno de los participantes falte, el que llegó primero quedará bloqueado.

3.6.2. Paso de mensajes asíncrono

El proceso que envía un mensaje no espera que esté listo el proceso que lo va a recibir. La implementación del lenguaje puede suspender al emisor hasta que el mensaje haya sido copiado para la transmisión; pero como no se ve reflejado en la semática, conceptualmente se dice que el emisor continúa inmediatamente después de enviar su mensaje.

[10] Suele llamarse rendezvous aunque este termino realmente se lo reserva para lo que Andrews [Andrews 91] llama rendezvous extendido.

3.5.3. Rendezvous y Remote Procedure Call (RPC)

Combinan características de monitores y paso de mensajes síncrono. Como los monitores, un proceso (módulo) exporta operaciones que pueden ser invocadas por otros procesos. Como con paso de mensajes síncrono, el proceso que realiza la llamada espera hasta que concluya la operación y posiblemente sean reportados algunos resultados. Así, una operación se comporta como un canal en dos direcciones, del invocador de la operación al proceso que la sirve y de éste al primero. La llamada puede ser atendida por un proceso que es creado exprofeso, en diferente procesador que el que aloja al proceso invocador (*RPC*); o por un proceso existente con el cual entabla una cita mediante alguna instrucción de entrada o aceptación que al recibir una invocación la procesa y devuelve resultados (*Rendezvous*[11]).

3.6. Tolerancia a fallas

Una de las ventajas que proponen los sistemas distribuidos frente a los centralizados es un grado mayor de confiabilidad y disponibilidad frente a fallas parciales, lo cual es lógico si se piensa que el control e información no dependen de un sólo elemento sino de más de uno en cooperación. Como en las sociedades humanas (ideales), si uno de sus miembros decae, los demás acuden a apoyarlo; así, en una red de computadoras, al fallar una máquina, alguna o todas las restantes pueden entrar a relevarla. La idea es clara, mas no el camino óptimo a seguir para concretarla. Este es un tema amplio en continua investigación.

3.7. Transparencia

Transparencia es probablemente un concepto tan viejo como la programación de computadoras. Surge del deseo de 'olvidar', abstraer u ocultar detalles subyacentes a una implementación, facilitando la manipulación de los detalles restantes [Pedregal 89].

Para Peter Wegner transparencia es un sinónimo cercano a encapsulación. Asevera que los Sistemas Distribuidos deben ofrecer los siguientes tipos de transparencia [Wegner 92]:

- Transparencia en casos de falla : permitiendo a los usuarios y programas de aplicación completar sus tareas, a pesar de fallas de hardware.
- Transparencia en el acceso a entidades locales y remotas : empleando operaciones idénticas.
- Transparencia en la localización de las entidades : posibilitando su acceso transparente.

[11] A este encuentro algunos autores llaman *Rendezvous Extendido*, para distinguirlo del que se da en el paso de mensajes síncrono, que llaman simplemente *Rendezvous*. Recuérdese que '*rendezvous*' significa cita, encuentro, reunión (en francés).

- **Transparencia en la concurrencia : habilitando la implementación de servicios concurrentes, sin el conocimiento de los clientes.**
- **Transparencia en la replicación : utilizando múltiples instancias de archivos y otros datos para el incremento de la confiabilidad y el desempeño.**
- **Transparencia en la migración : moviendo entidades dentro del sistema, sin afectar a los usuarios y sus programas de aplicación.**
- **Transparecia en el desempeño : autorizando la reconfiguración del sistema.**
- **Transparencia para escalar : permitiendo al sistema y aplicaciones expandirse en escala sin cambiar la estructura del sistema ni los algoritmos de las aplicaciones.**

¿ Es algún aspecto de los Sistemas Concurrentes y/o Distribuidos radicalmente preponderante sobre los demás ?

La respuesta parece una evasión al compromiso; pero es cierta : todo depende de la aplicación y de los recursos que se tengan. Por ejemplo, los sistemas en tiempo real no importa cuan flexibles sean; pero sí cuan predecibles; en una aplicación bancaria o de reservaciones aéreas el énfasis debe tener el servicio ininterrumpido de la red; en casos académicos (supuestamente no críticos en tiempo o dinero), se puede ser más exigente en expresividad. Entonces, ningún soporte para crear aplicaciones concurrentes y distribuidas tiene supremacía absoluta, su efectividad es relativa, según los problemas a los que vaya dirigido.

CAPITULO 3

COMBINANDO CONCURRENCIA Y DISTRIBUCION CON EL MODELO ORIENTADO A OBJETOS

Es reconocida la relevancia de los Sistemas Concurrentes y Distribuidos, también es notable la complejidad de los mismos. Por otra parte, el paradigma de la Orientación a Objetos ofrece persepectivas simplificadoras con garantías de calidad. Surge entonces la idea de visualizar los Sistemas Concurrentes y Distribuidos a través de las ideas de objetos, o visto de otro modo extender el modelo de Objetos para que soporten concurrencia y distribución.

El presente capítulo hace una breve sinópsis de las tendencias adoptadas para realizar el 'matrimonio' de estas vertientes, sus caraterísticas comunes, así como los principales conflictos entre sus conceptos.

De acuerdo al concepto de Sistemas Distribuidos como una variedad de los Sistemas Concurrentes, es importante primero aclarar los puntos referentes a Concurrencia. Para concretar un Sistema Distribuido OO los implementadores primero deben solucionar los problemas relacionados con los Sistemas Concurrentes OO. Por esta razón la primera parte de este capítulo se concentra en la combinación de la Concurrencia y el modelo OO, para después considerar la Distribución, donde la distribución física de componentes exige esfuerzos adicionales.

1. ¿ Por qué el deseo de combinar Concurrencia con el modelo OO ?

La Concurrencia es un esquema de construcción, una filosofía arquitectónica y si bien plantea una nueva persepectiva de modelaje de sistemas, realmente no puede ser considerada como un paradigma completo de programación. No es suficiente decir que hay procesos en comunicación y cooperación para resolver un fin común; se debe pensar que cada proceso es una componente con una tarea específica a resolver y la descripción de la componente así como la resolución de su tarea, independientemente del aspecto de concurrencia, debe modelarse de alguna manera. Por tanto, la concurrencia forzosamente debe combinarse con otro paradigma [Greiff 93]. En este sentido, no se dejaron esperar las propuestas para integrar en un solo modelo la Orientación a Objetos y la Concurrencia, esencialmente impulsadas por las semejanzas entre objetos y procesos, más la oferta general de la metodología OO, de facilitar la construcción de software complejo con resultados de alta calidad.

Las analogías entre las construcciones básicas de la programación OO y la programación concurrente : los Objetos y los procesos, o mejor aun, sus abstracciones respectivas, clase y tipo proceso, son evidentes. Ambas deben soportar [Meyer 93] :

+ Variables locales (atributos de una clase y variables de un tipo proceso).
+ Comportamiento encapsulado (un solo ciclo para un proceso o cualquier rutina en un Objeto)[1].
+ Datos persistentes, para almacenar sus valores entre activaciones sucesivas.
+ Restricciones rigurosas sobre el modo de intercambio de información.
+ Un mecanismo de comunicación usualmente basado en alguna forma de paso de mensajes.

Sin embargo, las similitudes iniciales no involucran todos los aspectos de la Orientación a Objetos (principalmente la herencia), ni está clara la forma en que los demás deben incorporarse para crear una aleación expresiva, potente y libre de conflictos entre las características de ambos modelos. Los puntos subsecuentes exponen los principales elementos a considerar, lamentablemente existe bastante disparidad en el criterio de los autores que investigan el tema.

[1]Se entiende que se trata del modelo de proceso con un cuerpo manifestando alguna actividad; se observa que en otros modelos, la actividad puede estar también en otras rutinas atendiendo varios requerimientos concurrentes.

2. ¿ Cómo obtener un lenguaje Concurrente Orientado a Objetos ?

El deseo de combinar Concurrencia y Orientación a Objetos ha dado lugar a diversas propuestas, que básicamente caen en alguno de los siguientes tres enfoques [Karaorman y Bruno 93] :

a. ***Diseño de un nuevo lenguaje Orientado a Objetos Concurrente.*** Los primeros sistemas tomaron esta directriz con concurrencia *'built-in'* . Por ejemplo Hybrid [Nierstrasz 87, 93] y ABCL/1 [Yonezawa 87].

b. ***Extensión de un lenguaje Orientado a Objetos existente.*** La mayoría de las extensiones combinan las siguientes técnicas :

- Heredan de clases especiales de concurrencia que el compilador modificado reconoce. Por ejemplo Eiffel// [Caromel 93] y PRESTO [Bershad, *et.al.* 88].
- Emplean palabras claves especiales, modificadores o técnicas de preprocesamiento para modificar o extender la sintaxis y la semántica del lenguaje. Por ejemplo CEiffel [Lohr 93]
- Extienden la sintaxis y semántica del lenguaje para soportar el paradigma general de concurrencia. Por ejemplo el modelo de Actor ACT++ [Kafura 89].

c. ***Diseño de una biblioteca.*** Es una solución atractiva porque no reemplaza el software existente, está influenciada por los trabajos recientes que buscan ante todo reusabilidad. Usa un lenguaje Orientado a Objetos existente y provee abstracciones de concurrencia a través de bibliotecas externas. Por ejemplo las clases desarrolladas por Karaorman y Bruno para extender Eiffel.

3. Objetos pasivos y Objetos activos

Los lenguajes OO secuenciales están basados en el empleo de ***objetos pasivos.*** Un *objeto pasivo* es activado (está en ejecución) cuando recibe el mensaje de otro objeto. En cualquier momento sólo existe un objeto activo en el sistema, dado que el objeto que envía un mensaje para solicitar un servicio (cliente) queda suspendido mientras el objeto invocado (servidor), deja su pasividad para cumplir la operación requerida, al concluirla y devolver un resultado se torna pasivo, activándose nuevamente el primer objeto [Bal, *et.al.* 89].

La concurrencia puede obtenerse extendiendo el modelo de objetos secuencial en atención a alguna o varias de las siguientes formas :

- Permitir a un objeto estar activo sin la necesidad de haber recibido el mensaje de otro objeto.
- Autorizar al objeto receptor continuar su ejecución después de regresar un resultado.
- Aprobar que el emisor de un mensaje proceda en paralelo con el receptor.
- Enviar mensajes a varios objetos a la vez.
- Capacitar al objeto servidor atender varios requerimientos a la vez.

La introducción de concurrencia en el modelo de Objetos promueve un nuevo concepto, el de ***objeto activo***. Los *objetos activos*, conocidos también como objetos proceso, son instancias que tienen asociados uno o más hilos de control (*threads*) que pueden estar activos o suspendidos. Uno de estos hilos puede atender la rutina conocida como *body* [America 90][2] o *live* [Caromel 93] que describe la secuencia de instrucciones que conforma el comportamiento del proceso, usualmente usado para controlar la recepción de mensajes. Después de recibir un mensaje, el hilo de control del *body* toma bajo su responsabilidad el invocar el método correspondiente a ese mensaje. En algunos lenguajes, el hilo de control del *body* se suspende durante el procesamiento del método, en otros, se ejecuta independientemente de los threads asociados al procesamiento de los demás métodos.

4. Manifestación de la concurrencia en un sistema OO

De acuerdo a los tipos de objetos que soporte el ambiente concurrente OO, el sistema puede presentar concurrencia inter-objeto, intra-objeto o ambas.

4.1. Concurrencia inter-objeto

Este tipo de concurrencia es simple debido a la independencia de los diferentes objetos, los cuales pueden ejecutarse al mismo tiempo, respetando la comunicación entre ellos a través de mecanismos que no comparten datos.

4.2. Concurrencia intra-objeto

La concurrencia intra-objeto o intra-nodo se da cuando dentro de un objeto se permite que varios hilos de control se ejecuten concurrentemente (o *quasi* concurrentemente).

5. Clasificación de los objetos de acuerdo a la concurrencia interna

Concurrencia interna es la que se manifiesta en el interior de un objeto. De acuerdo a ella se distinguen tres clases de objetos [Pedregal 89] [Nierstrasz 93] :

5.1. Objetos secuenciales

Tienen un solo hilo de control activo y puntos de entrada (con colas), donde las solicitudes esperan hasta que el proceso esté libre para servirlas. Nótese que si el hilo de control está suspendido -por ejemplo esperando por una condición de E/S- el proceso está bloqueado y ningún otro es servido hasta que la solicitud sea completada. Esta semántica presenta Ada en sus objetos activos [Wiener y Sincovec 90].

[2]Citada por [Meyer 93] al hacer una crítica referente a la distinción entre objetos pasivos y activos.

5.2. Objetos *Quasi*-paralelos

Estos objetos intercalan explícitamente múltiples hilos de control. Permiten que los hilos activos sean suspendidos en colas de condición para ser activados cuando algún otro hilo cause que la condición cambie. Cuando el hilo activo termina o es suspendido, un hilo de control en espera (ya sea en una cola de condición o en una cola de entrada -*entry queue*) es seleccionado para convertirse en activo. El resultado es que en un instante dado a lo sumo existe un solo hilo activo dentro del objeto. Los objetos de Hybrid [Nierstrasz 87] tienen este comportamiento.

5.3. Objetos paralelos

Los objetos concurrentes de este tipo no introducen restricciones (idealmente) sobre el número de hilos de control que pueden estar simultáneamente activos dentro de sí. Obviamente, deben implantar mecanismos de sincronización para el acceso y protección de sus datos internos y servicios. Argus [Liskov 88] es un ejemplo de este grupo.

6. Autonomía del objeto

Los objetos en un ambiente concurrente son responsables de proteger su estado interno en presencia de varios hilos de control, es deseable sea una labor automática liberando a los clientes de los servicios de la tarea de sincronizarse explícitamente entre ellos. Se reconocen básicamente tres versiones para abordar el problema según sea el modelo de objetos que soporte el lenguaje [Nierstrasz 93] :

a. ***Enfoque ortogonal.*** La sincronización es independiente de la encapsulación del objeto. Para evitar violaciones de la consistencia interna se utilizan mecanismos como seguros (*locks*) o semáforos.

b. ***Enfoque heterogéneo.*** Provee objetos activos y pasivos, adoleciendo de la falta de uniformidad, lo que impide intercambiar implementaciones de objetos de uno y otro tipo, limitando la reusabilidad y obligando a la posible duplicación de implementaciones de objetos similares; pero de naturaleza distinta (activa o pasiva).

c. ***Enfoque homogéneo.*** Todos los objetos son 'activos' y tienen control sobre la sincronización de solicitudes concurrentes. Esta modalidad puede incurrir en problemas de eficiencia.

7. Límite de los hilos de control con respecto a los objetos

La composición de los objetos activos y pasivos también puede ser analizada con respecto a su relación con los procesos. Hablando de proceso en el sentido del camino que recorre su *thread* o *threads* asociados para cumplir una acción[3].

La relación entre los procesos y los objetos caracteriza la composición de los objetos. Los procesos pueden estar separados y temporalmente acotados a los objetos que invocan o pueden estar acoplados y permanentemente acotados a los objetos en los cuales se ejecutan. Ambos enfoques corresponde a los modelos de objetos pasivos y activos, respectivamente [Chin y Chanson 91].

7.1. Modelo de objeto pasivo

Los procesos y los objetos son entidades separadas. Un proceso no está restringido ni acotado a un solo objeto. Un mismo proceso es usado para ejecutar todas las operaciones requeridas para satisfacer una acción. Este proceso no respeta los límites de los objetos porque durante su tiempo de vida, puede ejecutarse en el interior de todos los objetos que posean las operaciones invocadas. Un obstáculo en este modelo es el costo de mapeo del proceso en el espacio de direccionamiento de múltiples objetos.

7.2. Modelo de objeto activo

En el modelo de objeto activo con varios hilos de control, varios procesos servidores o trabajadores se crean y asignan para manejar los pedidos de operaciones. Cada proceso está restringido al objeto particular en el que fue creado. Siguiendo esta dirección, múltiples procesos pueden estar involucrados para resolver una acción y de acuerdo al número de éstos se distinguen dos opciones :

a. ***Variación estática.*** Crea un número fijo de procesos servidores cuando crea o activa a un objeto.

b. ***Variación dinámica.*** Los procesos servidores son creados dinámicamente por el objeto a medida que recibe requerimientos para sus operaciones. Las solicitudes no necesitan ser encoladas. Se visualiza el costo de la creación y destrucción contínua de procesos, cosa que es posible aliviarar, por ejemplo manteniendo una picina de procesos ociosos.

[3]Una operación puede invocar otra operación (posiblemente en otro objeto), ésta a su vez otra y así sucesivamente. Una cadena de invocaciones relacionadas se conoce como acción [Chin y Chanson 91].

8. La herencia en un ambiente concurrente

La herencia es el punto conflictivo en los Sistemas Concurrentes y/o distribuidos. Algunas aproximaciones la incluyen mediante estructuras complejas o inadecuadas (por lo menos conceptualmente), de ahí su sub-empleo o su aplicación en desmedro de otras características OO (como la encapsulación). Otras propuestas la eliminan radicalmente.

En muchos lenguajes OO concurrentes, los usuarios programan indicando explícitamente el conjunto de mensajes aceptables para cada objeto, con el propósito de implementar el comportamiento del objeto para que satisfaga las restricciones de sincronización. El código de sincronización es el término que se usa para referirse a la porción de código donde se controla el comportamiento del objeto con respecto a la sincronización. El código de sincronización debe ser consistente con las restricciones de sincronización del objeto; de otra manera, se producen errores semánticos durante la ejecución del programa. En virtud de programar el código de sincronización, cada lenguaje debe proveer un esquema de sincronización (primitivas o estructuras).

Desafortunadamente se ha observado que el código de sincronización no puede ser efectivamente heredado sin redefiniciones no triviales. Este conflicto es conocido como la ***anomalía de la herencia en lenguajes OO concurrentes*** [Matsuoka y Yonezawa 93][4] y es trascendente cuan supeditado está al esquema de sincronización del lenguaje utilizado; es decir, mientras que ciertas clases son heredadas con poca o ninguna redefinición en un lenguaje que provee algún esquema de sincronización, con el empleo de otro lenguaje que adopta un esquema de sincronización diferente, son precisas numerosas redefiniciones.

A continuación se dan algunos ejemplos de esquemas de sincronización y de las formas que adquiere la anomalía de la herencia para ciertos casos de estudio. La clase tomada como ilustración es un buffer.

8.1. Principales esquemas de sincronización en los Objetos

8.1.1. Cuerpos (bodies)

El objeto controla la recepción de mensajes en su función distinguida conocida como cuerpo. El control tiene generalmente la forma de la declaración select de Ada, por ejemplo :

[4]Revisar esta referencia para la consulta de detalles complementarios a los que se expongan en este trabajo.

```
class Buf  : Process
{    int  in, out, buf [SIZE];
  public:
     void put (int) {}  // Almacena un elemento en el buffer.
     int  get()   {}    // Extrae un elemento del buffer.

     void body ()
     {    loop
          {    select
               {    accept get () when (!(in == out))  start  get();
               or
                    accept put()      when (!(out == in + SIZE))  start  put();
               }
          }
     }
}
```

8.1.2. Conjuntos de aceptación

Los conjuntos de aceptación identifican los estados bajo los cuales pueden ser aceptados los requerimientos para los métodos del objeto, en el ejemplo de 'buffer' los estados distinguidos son *empty*, *partial* y *full*, cada uno contempla el conjunto de los métodos (realmente de sus identificadores) que pueden ser aceptados cuando el objeto se encuentra en ese estado. El esquema emplea una instrucción de conversión (*become*) para designar el conjunto de métodos a ser aceptados a continuación. Cuando estos conjuntos no son valores de primera clase el esquema se conoce como *Behavior Abstractions*, un esquema más avanzado que los modela como valores de primera clase es el de *Behavior Sets*; otro semejante en esencia a este último, es el esquema conocido como *First-Classing of Accept Sets* que basandose en la manipulación -con funciones y operadores- de lo que llama conjuntos habilitadores (*Enabled Sets*), redefine los conjuntos de aceptación.

A continuación se presenta el seudocódigo del esquema de abstracción de comportamiento :

```
class Buf  : Process
{    int  in, out, buf [SIZE];
  behavior:
     empty = {put};
     partial = {put, get};
     full    = {get};
  public:
     void Buf ()
     {  in = out = 0;
        become empty
     }
```

```
        void put (int)
        { // Almacena un elemento en el buffer.
          if (in == out + size)
              become full;
          else
              become partial;
        }

        int   proceso_get()
        { // Extrae un elemento del buffer.
          if (in == out)
              become empty;
          else
              become partial;
        }
    }
```

8.1.3. Métodos con guardias

Un esquema natural de sincronización es ligar un predicado a cada método como una guardia, convirtiéndolo en región crítica. Es una solución elegante; pero si no se toman precauciones de optimización se puede caer en ineficiencia. Con este esquema, el ejemplo de buffer queda así :

```
class Buf  : Process
{    int   in, out, buf [SIZE];
  public:
     void put (int)  when (in < out + SIZE)  {...}   // Almacena un elemento en el buffer.
     int  get()      when (in >= out +1)     {...}   // Extrae un elemento del buffer.
}
```

8.2. ¿ Por qué se da la anomalía de la herencia ?

Los sistemas OO en presencia de concurrencia se ven afectados por la anomalía de la herencia debido a los subsecuentes motivos :

8.2.1. Partición de los estados aceptables.

En un momento dado se dice que un objeto presenta algún 'estado'; en general, tiene un conjunto de 'estados'. Este conjunto puede ser dividido acorde a restricciones de sincronización; por ejemplo en el ejmplo de *buffer*, se distinguen tres estados : *empty*, *partial* y *full*, bajo los cuales sólo determinados métodos pueden ser aceptados. Si se añade un nuevo

método en una subclase, la división de estados debe ampliarse porque el nuevo método no fue tomado en cuenta en la clase padre. Por ejemplo si en la clase ***Buf2*** se define el método ***get2()*** que toma dos elementos en lugar de uno, particiona el estado *partial* en *partial_one* (un sólo elemento en el *buffer*) y *partial_other* (más de un elemento, habilitando la aceptación de una invocación de *get2()*).

Para los esquemas de aceptación este es un problema serio porque todos los métodos deben considerar el nuevo estado. Por otra parte las guardias en los métodos no introducen este conflicto porque son capaces de juzgar la aceptación o no de un mensaje directamente a partir del estado actual del objeto; así, el nuevo método independientemente estará controlado por su propia guardia, sin afectar los estados de aceptación de los demás.

8.2.2. Sensibilidad histórica de los estados aceptables.

Se aprecian dos vistas del estado de los objetos :

- ***Vista externa*** : donde el estado del objeto es capturado indirectamente a través del comportamiento observable externamente.

- ***Vista interna*** : donde el estado del objeto es capturado mediante la evaluación del estado de las variables en la implementación del objeto.

Las dos vistas no son idénticas, existen conjuntos de estados cuyos elementos son distinguibles bajo la vista externa e indistinguible bajo la interna, o viceversa. Con las guardias en los métodos, sólo los estados posteriores son distinguibles bajo la vista interna, debido a que la guardia involucra expresiones con valores booleanos, constantes y/o variables de instancia; luego, ciertas restricciones de sincronización no pueden ser especificadas con un conjunto dado de variables de instancia, se trata de la información histórica del objeto.

Si es necesaria la sensibilidad a hechos históricos, el estado del objeto bajo la vista interna debe ser refinado para unificar con la vista externa. Por ejemplo si en una subclase de ***Buf*** se tiene el método *gget()* que puede ser invocado únicamente si el método atendido previamente fue *get()*, se debe incluir una variable de instancia (llámese *after_put*) que guarde el rastro de la última invocación. Esto conlleva la modificación de todos los métodos para que hagan la asignación de un valor adecuado a la variable adicional. La anomalía persiste con el enfoque de los esquemas de conjuntos de aceptación.

8.2.3. Modificación de los estados aceptables.

Considérese la imposición de un seguro a los métodos de un objeto. Por ejemplo, si se crea una clase *Lock* heredera de ***Buf***, con dos nuevos métodos *lock()* y *unlock()*. Un objeto después de recibir el mensaje *lock()* suspende la recepción de los métodos *put()* y *get()* hasta que reciba el mensaje de *unlock()*. Los métodos de *Lock* son históricamente sensibles, de modo

similar a *gget()*, con la diferencia que *Lock* modifica el conjunto de estados bajo los cuales pueden ser invocados los métodos heredados de su clase ancestro. *Lock* introduce un grano más fino de distinción incluyendo una variable *locked*. No importa cual sea el estado del objeto (vacio, lleno o parcialmente lleno), la pregunta de si la instancia esta asegurada o no, sólo depende del estado de la variable *locked*, en este sentido, los métodos de *Lock* son ortogonalmente restrictivos.

La restricción ortogonal es una subclasificación importante y presupone en el esquema de métodos con guardias el cambio de las guardias para que los métodos se percaten de la nueva condición de sincronización; por su parte, los esquemas de conjuntos de aceptación, en su versión valores de primera clase, exhiben un buen comportamiento.

9. Aspectos OO en combinación con Sistemas Distribuidos

Los siguientes apartados enfocan los problemas a tratar cuando el Sistema Concurrente OO está distribuido.

9.1. La herencia y la distribución

Advertida la complejidad de los Sistemas Distribuidos es factible combatirla reutilizando y haciendo evolucionar las componentes ya creadas, mediante el uso de la herencia, como en el caso de cualquier sistema de software, sólo que en este caso su modificación paralela a su funcionamiento puede resultar indeseable. Ahora bien, con la versión de herencia dinámica, no sólo puede ser heredada la funcionalidad de los objetos, sino también podría ser modificada en cualquier momento, con el inconveniente de que grandes cantidades de código serían replicadas en diferentes sitios de la red. La repetición de código introduce los problemas de inconsistencia entre copias; pero permite que la esencia de un objeto no se encuentre en lugares distantes, evitando que la ejecución de un método involucre innecesariamente la ejecución de código disgregado en la red [Atkinson 91].

En consecuencia, la implementación completa del modelo OO en Sistemas Distribuidos es notoriamente difícil debido al soporte requerido para mecanismos de compartición. Específicamente, la herencia es el punto medular del problema, que vista como un mecanismo para compartir colapsa con la meta de independencia de piezas de un Sistema Distribuido. De acuerdo a Wegner, distribución y herencia son inconsistentes [Wegner 87][5]; aunque admite que las dimensiones esenciales de la Orientación a Objetos : encapsulación y reactividad con las de concurrencia y distribución[6] proveen un excelente marco para explorar el paradigma OO [Wegner 93].

Actualmente, los Sistemas Distribuidos antes que el modelo Orientado a Objetos han adoptado el modelo basado en Objetos [Chin y Chanson]. El propio Wegner [Wegner 93] afirma que los diseñadores de lenguajes deben estar conscientes de las propiedades de

[5]Considerado también por Atkinson[91] y Pedregal[87] en este contexto.
[6]Para él concurrencia y distribución no van indispensablemente ligadas.

tranparencia de los Sistemas Distribuidos y las de abstracción de la tecnología de software basada en componentes, dado que ambas son relevantes para el diseño de sistemas de cómputo futuros.

9.2. Localización y movilidad de objetos

9.2.1. Homogeneidad del modelo

Un sistema OO distribuido es responsable de manipular la invocación entre los objetos cooperantes. Cuando una acción realiza una invocación, el sistema debe localizar el objeto específico y desarrolar actividades pertinentes. Tomar o no un modelo homogéneo para todas las entidades es preponderante no sólo por aspectos de concurrencia; sino también de distribución sobre todo de transparencia en la localización y migración de los objetos : éstos pueden o no reflejar la distribución :

- Si no existe distinción entre objetos locales y remotos : los objetos grandes o pequeños tienen la misma semática sin importar si son locales o remotos.

- En una propuesa altemativa existen dos tipos de objetos (de diferentes tamaños) : uno corresponde al mecanismo normal de abstracción de datos (implementación de un TDA); y otro 'especial' que representa la abstracción de un nodo en un Sistema Distribuido, es decir un módulo que encapsula los recursos asociados a un nodo (constituye un nodo virtual).

9.2.2. Transparencia en la localización

La propiedad de tranparencia en la localización indica que un cliente no debe percatarse de la ubicación física del objeto que posee el servicio solicitado. Es el sistema quien debe determinar cual objeto debe recibir el mensaje y en que nodo físico reside. Con este fin, cada objeto debe tener un identificador que no debe cambiar durante su tiempo de vida, ni ser reusado por otro. El mecanismo para localizar un objeto debe ser suficientemente flexible como para permitir a los objetos migrar o moverse de un nodo a otro. Las propuestas observadas son :

a. **Esquema de codificación en el nombre.** Se codifica la localización del objeto como parte de su identificador. Es un esquema eficiente; pero impropio si se desea reubicar al objeto.

b. **Esquema de servidores de nombres distribuidos.** El sistema crea un grupo de objetos servidores de nombres que son mantenidos en cierto número (no necesariamente todas) de estaciones de trabajo[7].

[7]Con este nombre designamos a cualquier nodo físico, aunque no siempre se trate de una estación de trabajo.

c. **Esquema de *cache/broadcast*.** Mantiene un pequeño cache en cada estación de trabajo que graba las últimas localizaciones conocidas de un número de objetos remotos recientemente referenciados. Cuando no encuentra un objeto en su lista, difunde un mensaje preguntando por la localización actual del objeto requerido.

d. **Esquema con *forward location pointers*.** Un *forward location pointer* es una referencia usada para indicar la nueva ubicación de un objeto. Cuando un objeto es movilizado, se deja en la estación de trabajo de origen un apuntador de este tipo; para localizar un objeto que ha sido movido, sólo debe seguirse la cadena de apuntadores ligados a él. Este esquema puede ser utilizado con los enfoques anteriores. Por sí solo resulta inseguro, los apuntadores pueden perderse sobre todo en caso de fallas.

9.3. Casos de falla

Cuando un objeto lanza un mensaje a otro, la falla en la invocación admite dos formas :

a. **Falla existente.** Definida como la falla que ocurre antes de que la invocación inicie. Por ejemplo, cuando el objeto no puede ser localizado.

b. **Falla transitoria.** La falla que ocurre mientras que la invocación está siendo ejecutada.

Un sistema OO distribuido debe proveer mecanismos para que tanto los objetos clientes, como los objetos servidores del sistema, hagan un tratamiento adecuado de ambos tipos de fallas.

9.4. Confiabilidad de objetos

A medida que el número de componentes se acrecienta, la probabilidad de fallas aumenta, un Sistema Distribuido con numerosos Objetos diseminados por la red, debe ser capaz de detectar y recuperarse de las fallas de objetos individuales y de estaciones de trabajo. La confiabilidad puede obtenerse mediante los siguientes métodos :

a. **Recuperación de objetos.** En este modelo, se trata de rehabilitar al objeto que falla, tan pronto como sea posible. Se distinguen dos técnicas:

- Recuperación *roll-back*, el objeto es restablecido a su último estado consistente. Los procesos e invocaciones que estaban en desarrollo se pierden.
- Recuperación *roll-forward*, similar al anterior, con la diferencia que los procesos e invocaciones son reiniciadas permitiéndoseles continuar.

b. **Replicación de objetos.** Este esquema mantine copias de un objeto en diferentes estaciones de trabajo. Tiene las variantes dadas a continuación :

- Consentir unicamente la **replicación de objetos inmutables** (cuyo estado no puede ser modificado). Se evita problemas de inconsitencia y sincronización de accesos; pero está muy limitado al tipo de los objetos.

- **La replicación con copia primaria**, designa a un objeto como la réplica primaria, en ella se pueden hacer operaciones de modificación, otros objetos son tomados como las réplicas secundarias que sólo aceptan requerimientos de consulta. La copia primaria debe hacer del conocimiento de las secundarias cualquier modificación de sus estado. Una nueva subdivisión se aplica a este esquema :

  - Copia primaria estática, cuando ésta falla el cliente es prevenido para no hacer modificaciones hasta que el objeto esté recuperado.

  - Copia primaria dinámica, si el objeto de la copia primaria falla, una copia secundaria toma su lugar.

- La replicación con objetos *peer*, no designa a objetos primarios ni secundarios, todas las copias son semejantes y pueden atender solicitudes de cualquier tipo.

Un análisis comparativo de varios lenguaje distribuidos basados en objetos que cubren estos puntos y otros, puede encontrase en [Chin y Chanson 91], menos en [Bal, *et.al* 89] y [Pedregal 87]. Además de la literatura individual de cada lenguaje cuyas referencias algunas se dieron o darán en este trabajo y otras pueden ser consultadas en dichos artículos (principalmente los dos primeros).

CAPITULO 4

DESCRIPCION DE LOS ORIGENES Y DESARROLLO DEL TRABAJO

'Mecanismos de Interacción entre Procesos Distribuidos con la Metodología OO'

El trabajo desarrollado en esta tesis intenta beneficiarse de la combinación de los paradigmas de Orientación a Objetos y de Programación Concurrente y Distribuida, para crear un modelo jerárquico de componentes de software : los mecanismos de interacción entre procesos distribuidos. Se buscan formas estandar que sean independientes del soporte de implantación y de problemas de aplicación específicos, lo cual permite pensar que pueden ser adecuadamente reutilizables en varios contextos.

Este capítulo describe y sitúa el planteamiento seguido para concretar los objetivos del trabajo realizado.

Existen formas de interacción entre procesos que con frecuencia aparecen en aplicaciones de tipo concurrente y distribuido. El presente trabajo, ***Mecanismos de Interacción entre Procesos Distribuidos con la Metodología Orientada a Objetos***, al cual indistintamente se nombrará así o de manera resumida **MIPD-MOO**, busca distinguir estas formas y organizarlas en jerarquías de clases. Las clases encontradas representan modelos de procesos y mecanismos de comunicación y sincronización a diferentes niveles de complejidad. La construcción de nuevas clases implica el reuso de clases previamente definidas, siguiendo los lineamiento de la metodología OO. Estas clases son almacenables en una biblioteca de clases, que trata de brindar una gama, lo más amplia posible, de estandares de interacción interproceso, reutilizables por el programador de aplicaciones.

Previa a una semblanza general de los pasos seguidos en el desarrollo de este trabajo, se exponen ciertas características de algunos trabajos que de una forma u otra, son un marco de referencia para MIPD-MOO.

1. Trabajos relacionados

1.1. Trabajos considerados como antecedentes

El *proyecto de investigación de Sistemas Distribuidos y áreas afines*, a cargo del Dpto. de Ciencias de la Computación del IIMAS y la Facultad de Ciencias (ambas instituciones de la UNAM) [Albarrán, *et.al.* 89:90], es antecedente de varios trabajos, entre ellos, el propio MIPD-MOO. Este proyecto propuso crear un ambiente de programación distribuida, basado en una arquitectura que permita la expresión clara del paralelismo, sincronización y distribución, con el fin de facilitar el diseño, especificación y programación de aplicaciones distribuidas cooperantes y asíncronas, adaptando un conjunto de recursos heterogéneos existentes [Sánchez 90].

Antes de MIPD-MOO, se desarrolló un trabajo con parecidos objetivos generales, que basándose en la arquitectura de transputers y OCAM, añadió a los canales unidireccionales ofrecidos por el lenguaje, construcciones más complejas [Romero 93]. Este trabajo y el artículo y libro de Andrews[1] [Andrews 91] fueron considerados para crear varios de los mecanismos del trabajo que nos ocupa, con la principal diferencia que en MIPD-MOO, forman parte de un modelo amplio de mecanismo ligados jerárquicamente cuya raíz parte de los propios canales, que al igual que los mecanismos más complejos, son modelados con la metodología OO, haciendo principal énfasis en la reusabilidad.

Otros trabajos que no sirvieron como referencia de partida; pero *sí* como referencia de resultados porque coinciden en objetivos, entidades que manipulan o metodología empleada para fines similares son:

[1] Establece varios patrones de interacción, basados en paso de mensajes; pero definidos algorítmicamente en relación estrecha a ciertas aplicaciones, y no como entidades de interacción independientes.

1.2. Trabajos considerados relativos

1.2.1. *JOYCE+*

JOYCE+ [Franky 88] es una modificación del lenguaje *JOYCE* creado por Brinch Hansen [Hansen 87]. Utiliza un esquema de comunicación asincrónico[2] para programar Sistemas Distribuidos que van a operar sobre una red de comunicación de máquinas monoproceso y/o multiproceso. Permite programar procesos de manera estructurada, con interfaces bien definidas y comunicados indirectamente a través de puertos y canales, lo cual hace posible programar cada proceso sin tener que conocer nombres de otros procesos, adicionalmente, permite verificar desde compilación el uso correcto de las primitivas de comunicación.

JOYCE+ adolece de varias formas de canales. El que ofrece es unidireccional asíncrono, asociado a uno o más puertos de entrada y uno solo de salida. Cada puerto es un apuntador al canal. Cada uno de ellos es conceptualmente activo (cuando realmente se reduce a un 'buffer' con características de monitor). Los puertos se dice que son particulares a cada proceso; sin embargo, son pasados a otros procesos, rompiendo el encapsulamiento que podría tener el proceso. La creación de procesos es bastante restringida y propia de su diseño particular. Los canales tienen tipos asociados que deben coincidir en su declaración con el tipo de los puertos.

JOYCE+ se define como una extensión del lenguaje *PASCAL*. La exploración de otras metodologías como la Orientada a Objetos y la Orientada a Eventos se mencionan como perspectivas a futuro.

1.2.2. *Modelo de Proceso*

Con el avance de la tecnología de computación distribuida, se ha incrementado el interés por las **multiaplicaciones** (sistemas consistentes de múltiples aplicaciones corriendo en redes heterogéneas), un grupo de investigadores del *IBM T.J. Watson Research Center* presenta una estrategia para simplificar su programación. El enfoque adoptado se basa en integrar directamente a los lenguajes de programación (tarea dejada a sus implementadores), un modelo de programación distribuida de alto nivel, llamado el *Modelo de Proceso* (*Process Model*). Las aplicaciones distribuidas escritas en tales lenguajes son portables a diferentes ambientes y más cortas y simples, que cuando se realizan con enfoques convencionales.

El *Modelo de Proceso* define un modelo independiente del lenguaje y del sistema. Soporta la creación de procesos, interconección e interacción entre los mismos. Los principales conceptos manejados son :

[2]JOYCE emplea comunicación síncrona.

- proceso, puerto y liga,
- un conjunto de tipos de datos que pueden comunicarse y
- un mecanismo de definición de interfaz para especificar contratos entre los procesos (el contrato más simple establece el tipo de dato a ser enviado en un mensaje).

Un proceso puede estar escrito en cualquier lenguaje y debe poder leer, escribir y actualizar variables de cálculo, comunicarse (enviando o recibiendo sobre/de un puerto), conectarse estableciendo una liga entre un puerto de salida y un puerto de entrada. Los puertos son valores de primera clase. Los puertos son terminales de comunicación. Un puerto de entrada es una cola con datos que han sido enviados; pero no recibidos. Un puerto de salida es un liga a un puerto de entrada. Varios puertos de salida pueden apuntar al mismo puerto de entrada [Auerbach, *et.al.* 92].

Esta propuesta ofrece puertos de manera bastante parecida al de JOYCE+ aunque insertos en un modelo más amplio de comunicación y manejo de procesos. El modelo es ejemplificado en las implantaciones del lenguaje Hermes y Concert/C (una extensión de C) y no contempla aspectos de la metodología OO.

1.2.3. Paralelización transparente a través del reuso

J. M. Jézéquel [Jézéquel 93] afirma que los ambientes de software comercialmente disponibles para computadoras paralelas con memoria distribuida, consisten principalmente de bibliotecas de rutinas para manipular la comunicación entre procesos escritos en lenguajes secuenciales como *C* o *Fortran*; o bien recaen en compiladores paralelizadores semi-automáticos; ambos enfoques con sus propias desventajas. El autor propone la metodología OO para abordar el problema, piensa que el reuso de componentes de software podría ayudar a manejar la complejidad de la programación concurrente. Concentra esfuerzos en un modelo de paralización asociado a los datos. Parte de una clase donde define los datos organizados en una enumeración y determina, como una función diferida, la interfaz de alguna operación aplicable a cada uno de los datos (por ejemplo la raíz cuadrada) o a todos ellos (por ejemplo una reductora como la suma de sus valores). Las clases son planteadas secuencialmente; pero se mantienen para tener una implementación distribuida, combinándolas mediante herencia con clases que tienen el *know how* de la paralelización.

La idea de definir clases abstractas con operaciones a ser definidas por descendientes para aplicarlas indistintamente a una serie de datos es parecida a algunos mecanismos de MIPD-MOO (los explicados en el capítulo 6), sobre todo en el que se refiere al flujo entre *pipes*, donde el mecanismo completo de interacción comprende la organización de datos homogéneos en una serie a la cual se le debe aplicar una función de filtraje.

Jézéquel trabaja con datos numéricos en arreglos, matrices, etc., de hecho el modelo está dirigido a cálculos masivos en ese tipo de datos. MIPD-MOO es más flexible y de propósito general, funciona con mensajes que pueden adoptar cualquier forma (tipo de dato) y que indiscriminadamente pueden circular por una variedad de mecanismos, cada uno con

objetivos característicos que determinan la interfaz de las funciones a aplicar a la información. Cabe notar que bajo la misma forma de componentes de software reutilizables están no sólo los mecanismos para transformar los datos u obtener resultados, sino los mecanismos que facilitan la organización topológica de las componentes que van a trabajar los datos. Sería deseable ver una descripción más detallada de las clases de Jézequél para hacer distribuida la aplicación, quizas serían también combinables con la propuesta de los mecanismos de MIPD-MOO.

1.2.4. Creación de una arquitectuara reconfigurable

Saleh y Gautron [Saleh y Gautron] de Rank Xerox France y de la Universidad de París VI, exponen una aplicación que ha sido desarrollada para simular una máquina paralela altamente reconfigurable, donde los procesadores están físicamente cercanos y no comparten memoria ni ningún otro recurso, los canales son el único mecanismo provisto para intercambiar datos entre los procesadores. Conceptualmente un canal puede ser visto como dos ligas, la de entrada y la de salida. Existen dos clases de canales, uno en el que las dos ligas unen a los mismos procesos (por ejemplo, si la *liga 1* vincula al proceso A como fuente y a B como destino y la *liga 2*, une a B como fuente y a A como destino) y otros canales en los que los extremos no coinciden[3] (por ejemplo, si la *liga 1* toma a A y B, como fuente y destino respectivamente y la *liga 2*, a B y C como sus correspondientes fuente y destino)[4].

El trabajo hace un análisis comparativo de como el adoptar un lenguaje determinado (*Eiffel* o *C++*) afecta tempranamente el diseño del modelo, su lectura es interesante e ilustrativa para la cultura OO. En cuanto a los canales modelados realmente no forman una jerarquía de diferentes tipos de canales, aunque si exite una jerarquía al contruir el tipo de canales que presenta. Al ser éstos canales bidireccionales ofrecen la posibilidad de ver si los extremos son compartidos o no, lo cual determina si los atributos para almacenar los identificadores para los procesos conectados al canal deben ser únicos o duplicados, para lo que se explora la herencia repetida. Resulta útil ver que la obtención de canales bidireccionales puede resultar de la simple duplicación de los canales unidireccionales (los que pueden ser vistos como ligas). Aunque mantiene identificadores de los procesos conectados, no se observa un control de acceso al estilo de los canales con control de conexiones de MIPD-MOO.

2. La propuesta de MIPD-MOO

En capítulos previos se dijo que al realizar cómputo distribuido se cuenta con un conjunto de programas concurrentes en los cuales los procesos se comunican por el paso de mensajes [Andrews 91]. La función de un sistema de mensajes consiste en permitir que los procesos se comuniquen y sincronicen por el intercambio de información a través de canales y no por variables compartidas.

[3]La complicación de este tipo de canal parece innecesaria si se reemplaza por dos canales unidireccionales, uno de A a B y otro de B a C.

[4]El artículo no muestra identificadores que claramente distingan respecto a quien son fuente y destino.

Como las aplicaciones a las que va dirigido este trabajo son de tipo distribuido, es preciso identificar el elemento principal de ese sistema de mensajes : el canal. Un **canal** es un dispositivo de comunicación interproceso accesado básicamente por dos primitivas : una de aceptación y otra de entrega de mensajes. Puede implantarse de diferente modo físico y lógico, el presente trabajo se ocupa del segundo, considerando al canal como una abstracción de un enlace físico, que provee una ruta de comunicación entre los procesos. Estos se distinguen por su habilidad de envío de mensajes (procesos emisores) o recepción de ellos (procesos receptores), empleando las facilidades del canal (o canales) que los comunica(n).

Todas las notaciones de programación basadas en el paso de mensajes otorgan canales y primitivas de algún tipo, pudiendo variar en :

- la forma en que los canales son provistos y nombrados,
- el modo en que los canales son usados y
- la manera en que la comunicación es sincronizada.

Por ejemplo:

a. Los canales :

- Son globales a procesos o directamente asociados con ellos.
- Permiten el flujo de información en una o dos direcciones.
- Tienen capacidad de almacenamiento : cero, finita, infinita.
- Conectan uno o varios emisores con uno o varios receptores.
- Tienen un tipo asociado o no tienen restricciones.

b. Los mensajes :

- Tienen o no un tipo asociado.
- Son de tamaño fijo o variable.
- Varían en la información que transportan.

c. La comunicación y sincronización :

- Bloqueante,
- No bloqueante.

Combinando algunas de las opciones propuestas surgen varios modelos (no ortogonales), cada cual con su respectiva importancia porque la versión de canales y primitivas elegidos para cada uno forman mecanismos de interacción adecuados para resolver cierto tipo de problemas.

La metodología OO allana la elaboración de modelos flexibles a cambios y extensiones, por lo que se piensa que utilizando sus virtudes se pueden construir mecanismos de interacción partiendo de esquemas básicos para llegar, con especializaciones o generalizaciones, a los más complicados. Así, los mecanismos presentan varios niveles de complejidad, brindando una colección a la cual los programadores de aplicaciones de diferentes ámbitos, pueden recurrir para elegir una instancia dada que ajustarán a sus necesidades con el menor esfuerzo posible, también gracias a las técnicas OO.

2.1. Niveles comprendidos en el trabajo MIPD-MOO

La estructura del trabajo sigue la jerarquía de complejidad de las clases que intervienen en los distintos modelos de interacción, según el siguiente desarrollo :

2.1.1. PRIMER NIVEL

Se tomó como plataformas de sincronización y comunicación el paso de mensajes síncrono y el asíncrono. Sobre ellas se construyó una jerarquía que inicia con el reconocimiento de las clases básicas más relevantes. En este nivel debe destacarse a canal que no es una clase sino toda un grupo jerarquizado de clases, dependiente de la sincronía, del control de conexiones o del número de proceso conectados a sus extremos.

2.1.2. SEGUNDO NIVEL

Una vez definidas las clases básicas que permiten definir la topología de las conexiones (a nivel lógico), modelamos clases con comportamientos específicos como ser 'pipe', 'buffer', difusor, etc.

2.1.3. TERCER NIVEL

Si bien las clases anteriores son construidas vislumbrando determinados patrones de interacción, recién en esta etapa se las integra con otras clases auxiliares para formar verdaderos modelos de interacción de cierta naturaleza, llámense flujo en 'pipes', productor-consumidor, difusión de mensajes, etc.

Los modelos de interacción pueden considerarse aplicaciones que reutilizan las clases más elementales; sin embargo siendo clases altamente suceptibles a ser reutilizadas a su vez por otras aplicaciones preferimos pensar en ellas como partes de una biblioteca general de clases reusables.

2.1.4. CUARTO NIVEL

En este nivel se observa la utilidad de las clases predefinidas en la creación de ejemplos de aplicación propios de programación concurrente y distribuida.

Lograr modelar y programar todas las formas de interacción no es factible ni lógico. Nosotros sólo adoptamos las más notables, mostramos el potencial de las clases reutilizables y dejamos que la biblioteca formada con ellas se mantega abierta a nuevas inclusiones.

A continuación sintetizando lo expuesto, se despliega un cuadro que exhibe el curso del trabajo. Algunos ejemplos realmente no se han incluido en el modelo desarrollado (sobre todo del último nivel); pero se agregan en el cuadro para dar una mayor orientación de la dirección que sigue el trabajo. Se tomaron ejemplos de aplicación ilustrativos aunque no extensos como pueden ser ciertas aplicaciones reales citadas. Por otro lado, algunos mecanismos no se citan en el cuadro; sin embargo son expuestos en los capítulos siguientes.

MECANISMOS DE INTERACCION Y SUS APLICACIONES

PROBLEMAS DE APLICACION (4to. nivel) :

Ordenamiento en paralelo
Reconocimiento de números primos
Servidores de archivo
Topología de una red
Detección de terminación
Semáforos distribuidos
Mantenimiento de BD replicadas
Rastreo de Mensajes

MODELOS DE INTERACCION (3er. nivel) :

Flujo en 'pipes'
Productor - consumidor
'Broadcast'
Cliente - servidor
'Heartbeat'
Bolsa de trabajo

CLASES ESPECIALES (2do. nivel) :

'Pipe'	Productor	Puente	Cliente	Difusor
'Peer'	Consumidor	'Buffer'	Servidor	Sumidero

CLASES BASICAS (1er. nivel) :

Emisor Canal Mensaje Receptor

Modelos de sincronización y comunicación :

Paso de mensajes asíncrono Paso de mensajes síncrono

Los diagramas de clases responden al método de Booch, método de análisis y diseño OO que mayormente influenció para la conceptualización de las clases[5].

El seudocódigo empleado para expresar el modelo de los mecanismos de MIPD-MOO, se adapta a la sintaxis y semántica del lenguaje C++Concurrente, el cual fue utilizado para pruebas experimentales del modelo. A continuación se incluye la descripción de sus principales aspectos.

2.2. C++ Concurrente (C++C)

C++C [Olmedo 93] es una extensión al lenguaje Turbo C++ (v.1.0 - v. 2.0, lo que presupone las limitaciones de estas versiones como ser la carencia de genericidad a través de *templates*), para proporcionarle aspectos de concurrencia en un ambiente monoprocesador (el de una computadora personal).

2.2.1. Caracteristicas relevantes

- Abstracción de datos mediante clases (C++).
- Creación y destrucción estática y dinámica de procesos y de corrutinas (extensión).
- Topología estática y dinámica de procesos (extensión).
- Programación en el estilo de Proceso Distribuido (extensión).
- Ejecución no determinista de programas (extensión).
- Comunicación síncrona bidireccional (consecuencia del modelo).
- Comunicación por canales (extensión).
- Compilación separada de módulos y facilidades de depuración (ambiente).
- Caracterización del sistema operativo como un proceso prestador de servicios (extensión).
- No incluye :
  - Manejo de excepciones.
  - Manejo de interrupciones.
  - Métodos de verificación de especificaciones.

2.2.2. Palabras reservadas

C++C añade las siguientes :

process **nullprocess** **select** **when**

[5]La explicación extensa de este método se encuentra en [Booch 91, 94], una versión resumida en [Oktaba 93] y la descripción de una herramienta para su empleo en [Rational Rose 92, 93].

2.2.3. Entidades en un programa en C++C

Un programa en C++C presenta objetos pasivos y activos.

- ***Objeto*** = entidad pasiva = objeto pasivo
  = objeto activo con actividad nula.
- ***Proceso*** = entidad activa = objeto activo.

Un programa concurrente consiste de varios procesos que están corriendo simultáneamente y que pueden durar indefinidamente. Cada proceso es una instancia de alguna clase que hereda de la clase distinguida *Process*, posee sus propias variables locales y no requiere de variables comunes para comunicarse.

2.2.4. Sincronización y comunicación entre procesos

Los procesos se sincronizan por medio de mecanismos de varios niveles de abstracción.

a. Operaciones básicas de sincronización :

```
suspend ();
transfer ();
```

b. Instrucción de sincronización *select* (al estilo del comando custodiado de Dijkstra) :

```
select when (cond) estatutos;
```

o bien :

```
select
{   when (cond_1) estatutos_1;
    when (cond_2) estatutos_2;
    ....
    when (cond_n) estatutos_n;
}
```

c. Funciones públicas que ofrece un proceso (mecanismo también de comunicación) :

```
proceso.servicio (parámetros actuales)
proceso->servicio (parámetros actuales)
```

2.2.5. Construcción y destrucción de procesos

- Creación estática : cuando se declara una variable.
- Creación dinámica : con operador **new**.
- Destrucción : cuando el control alcanza el final del bloque que lo define o por la aplicación del operador **delete**.

2.2.6. Procesos y corrutinas

La diferencia radica en la forma como se transfiere el control :

- La corrutina transfiere (**transfer()**) el control directamente a otra corrutina y, por esta razón, debe identificarla a través de un apuntador.
- Un proceso renuncia el control (**suspend()**) y es el administrador del núcleo de concurrencia el que determina el siguiente proceso a ejecutar.

2.2.7. Cambio de contexto

Comprende las acciones necesarias para suspender al proceso activo, y para reactivar otro que está en condiciones de reanudar su ejecución. Las acciones son :

a. Guardar el ambiente de la ejecución del proceso activo en su pila.
b. Guardar los apuntadores a la pila del proceso en su descriptor.
c. Elegir otro proceso que pueda reanudar su ejecución.
d. Restaurar los apuntadores a la pila para el proceso elegido.
e. Finalmente, restaurar el ambiente de la ejecución del proceso elegido.

Por lo general, en un cambio de contexto existe más de un proceso que puede recibir el procesador por lo que se forman en una cola de espera predefinida por el sistema conocida como **readyQueue**. Para resolver los conflictos se sigue una política de asignación -la de *round-robin*.

2.2.8. Especificación de procesos

Establece la estructura interna e interfaz de los procesos mediante la declaración de variables y funciones componentes. La especificación de cualquier clase debe incluir al menos un constructor. Por ejemplo, para un escritor :

```
class Writer : Process
{   int c;
  public:
    Writer (char*, int);
};
```

2.2.9. Definición de procesos

Un proceso se define implantando las funciones declaradas en su especificación. La parte primordial de la definición la constituye la implantación de la función constructora, debido a que :

- El comportamiento, generalmente no determinístico, lo establece el texto de esta función.
- Diferentes funciones constructoras definen comportamientos diferentes en procesos de la misma clase.
- La duración del proceso es la misma que la del bloque que define la función constructora.

Por ejemplo el comportamiento del escritor especificado previamente está descrito mediante la siguiente función constructora :

```
// Proceso Writer con tamaño de pila y prioridad establecidos exprofesamente,
// sobreencimando los valores default.
Writer::Writer (char* s, int n) : Proceso(128,100)
{    cout << "Proceso" << s << "inicia su actividad\n";
     for (c=0; c<n; c++)
     {  cout << s << c;
        suspend ();
     }
     cout << "Proceso" << s << "termina su actividad\n";
}
```

2.2.10. El proceso *Nulo*

Es un proceso predefinido cuyo propósito es el de simplificar tanto el mecanismo de principio y fin de la actividad concurrente como el algoritmo de asignación del procesador. Todo programa concurrente debe incluir la declaración de un objeto de este tipo de proceso.

2.2.11. Declaración de objetos activos

Cuando se declara un objeto de cierta clase de procesos, se crea automáticamente el proceso, aunque esto no significa que su actividad comience de inmediato. Sólo después de la creación del proceso nulo, los procesos declarados antes pueden entrar en competencia por el uso del procesador. Por ejemplo :

```
// Lanzamiento de varios escritores concurrentes
main (void)
{    Writer a ("Escritor A:",4), b("Escritor B:",2), Writer c("Escritor C:",3);
     Null os;      // Creación del proceso nulo.
}
```

CAPITULO 5

JERARQUIAS DE CLASES BASICAS PARA EL SOPORTE DE LA COMUNICACION Y SINCRONIZACION POR PASO DE MENSAJES

Los procesos que conforman los programas distribuidos desarrollan su actividad de forma independiente, mas no aislada. Su comportamiento, suele depender de la información que intercambian entre sí. La información, en forma de mensajes, circula entre los procesos que desean comunicarse a través de los canales que los conectan.

En este capítulo se presentan modelos jerárquicos para los mensajes, los canales y los procesos emisores y receptores. Las clases construidas permiten crear la topología de una red lógica de comunicaciones entre procesos distribuidos, dotados de primitivas básicas de envío y recepción de mensajes, mediante canales. Al final se incluyen algunos ejemplos que exhiben su aplicación. En el próximo capítulo se reutilizan estos modelos para crear mecanismos de interacción más complejos.

1. DESCRIPCION Y MODELADO DE LOS ELEMENTOS QUE INTERVIENEN EN LA COMUNICACION Y SINCRONIZACION POR PASO DE MENSAJES

1.1. Proceso *emisor/receptor* general

Sean dos procesos, para que el primero transmita información al segundo, es necesario que éste sea un receptor de mensajes y aquel un emisor. La capacidad de envío o recepción distingue a ambos procesos; sin embargo, las aplicaciones indican que generalmente, cada proceso requiere ser tanto emisor como receptor, entonces, es deseable dotar a cada proceso de la capacidad de envío y recepción. Conoceremos a cada proceso de este tipo bajo el nombre de proceso ***emisor/receptor general***. Sus características, para el modelo construido, son las siguientes :

a. ***Emisor/receptor general*** es un proceso y por tanto, un ente activo.

b. Un proceso ***emisor/receptor general*** puede enviar y recibir mensajes, es tanto un emisor como un receptor, aunque en tiempos distintos y para diferentes canales, es decir :

c. ningún proceso ***emisor/receptor general*** es fuente y destino para el mismo canal.

1.1.1. Modelo OO del proceso *emisor/receptor general*

Se concibe que todos, o al menos la mayoría de los procesos en una aplicación distribuida, son *emisores/receptores*, envían y reciben mensajes; pero no como actividad final, por el contrario, estas funciones sólo son parte de un comportamiento tan complejo y variado como sea la aplicación en la que se encuentran. La metodología OO permite ahorrar trabajo a los programadores de aplicaciones, tomando como algo dado las capacidades de comunicación comunes a cualquier proceso distribuido, para concretarse a elaborar la parte propia de la aplicación. Esto se logra de un modo natural si se dispone de una clase con habilidades de un proceso *emisor/receptor*, que sirviendo como base permite ser enriquecida con nuevas aptitudes según la conveniencia de la aplicación. Esa clase se denomina ***EmiRec*** y los procesos de las aplicaciones son sus especializaciones.

La Fig. 5.1 muestra la clase *EmiRec* y su relación con otras clases que se explicarán conforme sea el avance de este capítulo.

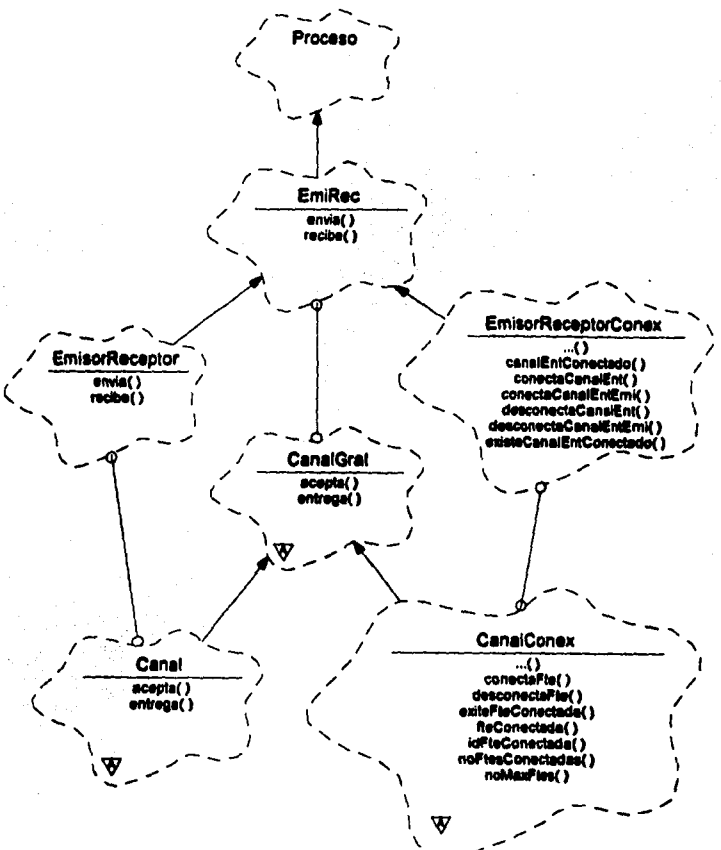

Fig. 5.1 Jerarquía de clases para procesos emisor/receptor de mensajes y su relación con algunas clases de canales.

En cuanto a caracteríticas teóricas, la clase *EmiRec* cumple los siguientes requerimientos :

a. *EmiRec* es una clase heredera de la clase *Process* (*Process* otorga comportamiento concurrente a sus descendientes); por tanto, los objetos creados a partir de ella son objetos activos (procesos).

b. La aportación de la clase *EmiRec* a cualquiera de sus descendientes es la noción del envío y/o la recepción de mensajes, a través de canales.

c. A este nivel, el conducto o canal por el que viajan los mensajes es de tipo general, se trata de un *CanalGral*[1] (ver Fig. 5.1 o siguientes).

d. El envío de un mensaje significa entregar dicho mensaje al canal que lo transmitirá. No es conveniente que se transmita la dirección del mensaje puesto que nada garantiza que el proceso que lo envía no lo modifique (después de entregarlo al canal), provocando un efecto colateral que perjudique al proceso receptor. Por esta razón, lo que se transmite es una copia del mensaje.

e. El mensaje que se envía y/o recibe puede ser de diferentes tipos, dependiendo de la aplicación.

   ¿ Cómo obtener la copia de cualquier mensaje, a este nivel de generalidad ?

   Gracias a las propiedades de polimorfismo y liga dinámica del modelo de Objetos, cada mensaje lleva el conocimiento de la creación de su propia copia en la operación *creaCopia(...)*, la cual polimórficamente se adecua al tipo de mensaje que en ese momento se esté transmitiendo[2]. La instancia de *EmiRec* sólo necesita invocar esta operación olvidando el tipo del mensaje actual.

f. Recibir un mensaje se reduce a la solicitar al canal de comunicación la entrega del mismo.

g. Nótese que el atributo protegido *'msjCom'* actúa como puerto de entrada/salida. En él se deposita el mensaje a ser inmediatamente transmitido, en una operación de envío, o el mensaje que fue recientemente recibido, en una operación de recepción.

h. El éxito o error en la ejecución de las operaciones *envia(...)* y *recibe(...)* de *EmiRec*, se constata a través del valor que devuelven (de tipo ESTADO), mismo que es propagación del valor devuelto por las operaciones *acepta(...)* y *entrega(...)*, pertenecientes al canal de transmisión.

---

[1] Por el momento, es suficiente saber que cualquier canal ofrece las operaciones de *acepta()* y *entrega()*.
[2] Ver más adelante en la descripción de clase 'Mensaje' y sus descendientes.

i. La clase ***EmiRec*** es el primer nivel de una jerarquía de *emisores/receptores*, para diferenciar sus objetos de los de sus descendientes, ***EmiRec***, la clase superior, posee un atributo de identificación que es heredable y mantendrá el valor apropiado de cada objeto (ésto podría evitarse si se contará con alguna habilidad similar ya provista implícitamente en el sistema de control de tipos del lenguaje).

1.1.2. Seudocódigo de la clase *EmiRec*

```
class EmiRec : public Process
{
  private:
    friend class CanalGral;
      Mensaje          *msjCom;
  protected:
      ID_EMI_REC       identificador;

      ESTADO           envia (Mensaje*, CanalGral*);
      ESTADO             recibe (Mensaje*&, CanalGral*);
      void             escribe (Mensaje* m)                    { msjCom = m; }
      Mensaje*           lee ()                                { return msjCom; }
      virtual  void    asignaId (ID_EMI_REC id = EMI_REC)      { identificador = id; }
  public:
                       EmiRec (ID_EMI_REC id=EMI_REC)          { identificador = id; }
      ID_EMI_REC       leeId ()                                { return identificador; }
};
```

envia(Mensaje* m, CanalGral* c) crea una copia del mensaje a transmitir *'m'* y lo deposita en el puerto *'msjCom'* del proceso *EmiRec* actual -es quien ha invocado la presente operación-, después notifica al canal *'c'* para que acepte su mensaje y lo transmita.

```
ESTADO EmiRec::envia(Mensaje* m, CanalGral* c)
{
  msjCom = m->creaCopia(m);[3]
  return (c->acepta(this));
}
```

recibe(Mensaje* m, CanalGral* c) solicita al canal *'c'* que le entregue un mensaje; el canal, en caso de éxito en su operación de entrega, deja el mensaje en el puerto de salida/entrada *'msjCom'* del proceso receptor actual. Al concluir la operación, *'m'* queda con el valor de *'msjCom'*.

[3]En este caso la copia del mensaje se saca antes de depositarlo en el canal, una decisión alterna es crearla en el propio canal.

```
ESTADO EmiRec::recibe (Mensaje*& m, CanalGral* c)
{
  ESTADO edoOp;

  edoOp = c->entrega(this);
  if (edoOp == EXITO)
    m = majCom;
  return edoOp;
}
```

1.2. Jerarquía de *mensajes*

Los mensajes son los elementos de información que fluyen de los procesos emisores a los receptores. Cada programa de aplicación manipula mensajes de diferente naturaleza, principalemente en cuanto a su contenido, desde los más simples a los más complejos, por ejemplo de tipo caracter, entero, sonido, gráfico, etc.

Cualquier *mensaje* puede caracterizarse por :

a. Un identificador.
b. Datos de control (como la prioridad).
c. Un contenido.

1.2.1. Modelo OO de los mensajes

La forma como se transmiten los mensajes es independiente de su contenido, por ello se introduce la clase general *Mensaje*, que permite modelar las clases necesarias para la comunicación interproceso, sin hacer consideraciones sobre el tipo del contenido de los mensajes, siendo cada aplicación la que de acuerdo a sus necesidades especializa la clase general. La especialización se logra a través de la herencia, las nuevas clases son descendientes de *Mensaje*. Por ejemplo, clases derivadas de *Mensaje* pueden ser : *MensajeEntero*, *MensajeControl*, *MensajeCadena* , *MesajeDibujo*, etc., algunos de estos posibles ejemplos se pueden observar en la Fig. 5.2.

A continuación las características de la clase *Mensaje* :

a. *Mensaje* es una clase abstracta[4] que determina la interfaz que deben cumplir los mensajes que se derivan de ella y que son suceptibles a ser transferidos, esta interfaz permite modelar de forma general las clases que intervienen en la transmisión.

b. La función *creaCopia(...)* es parte de la interfaz de la clase y es una función virtual pura, lo que significa que necesariamente debe ser definida en todos los descendientes de *Mensaje*. Crea una copia del mensaje en transmisión, ajustándose polimórficamente al tipo de la instancia.

[4]Clase pura, en el dominio de C++.

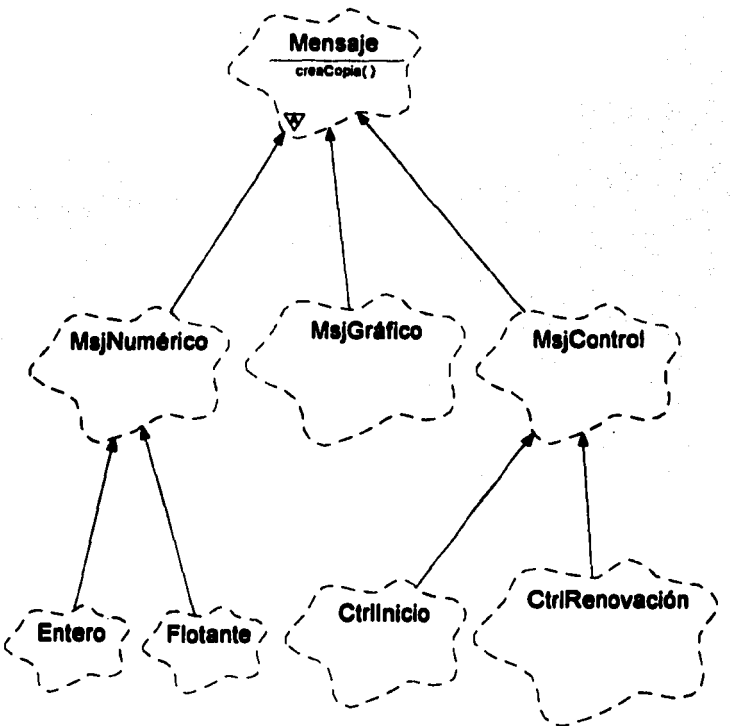

Fig. 5.2 Jerarquía de clases para mensajes.

c. El atributo identificador es heredable y almacena el tipo del mensaje para cualquier instancia *Mensaje* [5] o sus clases derivadas.

d. *Mensaje* tiene un atributo de prioridad, útil para establecer el orden de entrega de los mensajes enviado por un canal asíncrono, y en general en cualquier circunstancia, para implementar diferentes políticas en el manejo de mensajes.

1.2.2. Seudocódigo de la clase *Mensaje* y algunas de sus clases derivadas

Mensaje es la clase superior de la jerarquía de mensajes.

```
class Mensaje
{ protected:
    friend class Nodo;          // Amistad necesaria para poder encolar
    friend class Cola;          // los mensajes [6].
    unsigned prioridad;
    ID_MSJ identificador;
  public:
                Mensaje (unsigned p=PRIO_MSJ, ID_MSJ id = MSJ_INDEF)
                                        { prioridad = p; identificador =id; }
    virtual Mensaje* creaCopia (Mensaje*) = 0;      // Función a definirse en descendientes.
    ID_MSJ      leeId (void)            { return identificador; }
    virtual void    asignaId (ID_MSJ id = MSJ_INDEF)  { identificador = id; }
    virtual void    vista (void);                     // Despliegue de características.
};
```

La clase *MensajeEntero* redefine algunas funciones de *Mesaje* y añade un contenido de tipo entero, así como operaciones que lo manipulan.

```
class MensajeEntero : public Mensaje
{ private:
    int contenido;
  public:
            MensajeEntero (unsigned p=PRIO_MSJ, ID_MSJ id=MSJ_ENTERO) :
                        Mensaje (p,id) {}
    Mensaje*  creaCopia (Mensaje*);         // Definición de la función pura.
    void      copia (MensajeEntero m)       { contenido = m.contenido; }
    void      escribeContenido (int cont)   { contenido = cont; }
    int       leeContenido ()               { return contenido; }
    BOOL      mismoContenido (MensajeEntero m)  { return contenido == m.contenido; }
    void      vista (void);                 // Redefinición de despliegue.
};
```

[5] Como en el caso de *EmiRec* y otras clases explicadas más adelante, este identificador de Mensaje podría omitirse, si se trabajase con alguna versión de lenguaje OO que tenga implícito un identificador que acompañe a las instancias de las clases.

[6] No necesario dependiendo de las facilidades que ofrezca el lenguaje de implementación.

```
Mensaje* MensajeEntero::creaCopia (Mensaje* m)
{
  MensajeEntero *mm;

  mm = new MensajeEntero;
  mm->identificador = ((MensajeEntero*)m)->identificador;
  mm->prioridad     = ((MensajeEntero*)m)->prioridad;
  mm->contenido     = ((MensajeEntero*)m)->contenido;
  return mm;
}
```

La clase *MensajeControl* es otro ejemplo de como se puede especializar la clase superior *Mensaje*, en este caso, el mensaje carece de contenido y sirve sólo para señalizaciones, por ejemplo el inicio o fin de un ciclo de envíos. Puede a su vez especilizarse en varios tipos de control.

```
class MensajeControl: public Mensaje
{
  public:
    MensajeControl (unsigned p=PRIO_MSJ, ID_MSJ id=MSJ_CONTROL) : Mensaje (p,id) {}
    Mensaje* creaCopia (Mensaje*);
};


Mensaje* MensajeControl::creaCopia (Mensaje* m)
{
  MensajeControl *mm;

  mm = new MensajeControl;
  mm->identificador = ((MensajeControl*)m)->identificador;
  mm->prioridad     = ((MensajeControl*)m)->prioridad;
  return mm;
}
```

1.3. Canales de comunicación en su forma general

Los canales son la abstracción de la red de comunicación física, proveen la vía de comunicación entre los procesos. En un ambiente donde se usa el paso de mensajes (distribuido), los canales son típicamente los únicos objetos que se comparten [Andrews 91].

Los canales pueden ser clasificados desde varias perspectivas y ofrecer diferentes facilidades[7], dando lugar a diversos tipos de ellos, el presente trabajo abarca varios de ellos buscando organizarlos en un modelo jerárquico consistente (ver la Fig. 5.3).

[7] Ver capítulo anterior.

Un ***canal general*** en el modelo desarrollado es un mecanismo que :

a. Permite el paso de la información en una dirección (unidireccional).

b. No puede tener, en igual tiempo, como proceso fuente (emisor) y destino (receptor) al mismo proceso.

c. Posee primitivas de acceso para la transmisión de mensajes.

1.3.1. Modelo OO de los *canales generales*

El canal de características generales se traduce en una clase denominada ***CanalGral*** (visible en la jeraquía de clases de la Fig. 5.3), descrita de la siguiente manera :

a. *CanalGral* es una clase que origina objetos pasivos.

b. Posee un atributo de identificación que sirve para diferenciar las distintas instancias.

c. Necesita tener acceso a los puertos (*'msjCom'*), de los procesos *EmiRec* que actúan como emisor y receptor conectados a él, ésto se logra declarando a ***CanalGral 'friend' de la clase EmiRec***. Adicionalmente se precisa que las clases derivadas de *CanalGral* también estén autorizadas.

   ¿ Cómo pueden tener acceso a 'EmiRec' las clases derivadas de 'CanalGral' si la relación de amistad no es heredable ?

   Se fijan funciones que usan la parte protegida de *EmiRec*, como protegidas en la clase *CanalGral*, de tal suerte que indirectamente los descendientes de *CanalGral* pueden disponer de la porción protegida de *EmiRec*.

d. Aprueba que su sección protegida sea a su vez empleada por la clase *EmiRec*, declarándola amiga.

e. La definición de sus funciones *acepta(...)* y *entrega(...)* se relega a las clases derivadas, que las establecerán de acuerdo a criterios propios de sincronización.

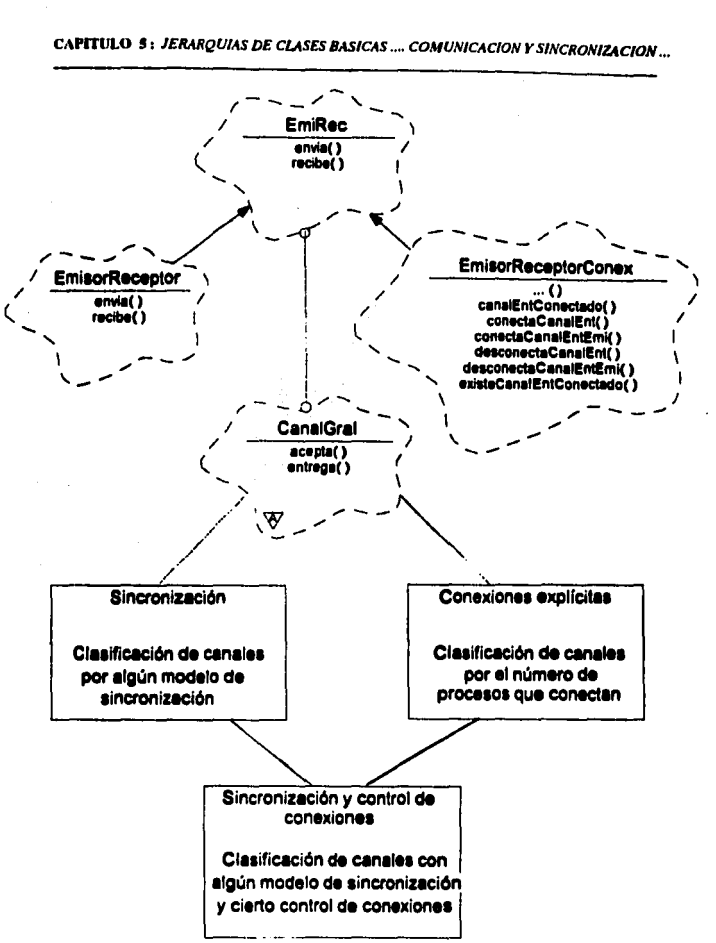

Fig. 5.3 Jerarquía de clases para procesos emisor/receptor de mensajes y las principales categorías de canales.

1.3.2. Seudocódigo de la clase *CanalGral*

```
clase CanalGral : public Proceso
{
  protected:
    friend class EmiRec; [8]
      ID_CANAL        identificador;

      virtual ESTADO  acepta (EmiRec*)  = 0;
      virtual ESTADO  entrega (EmiRec*) = 0;
      Mensaje*        leeMsjCom (EmiRec* fte)              { return fte->lee(); } [9]
      void            escribeMsjCom (EmiRec* dest, Mensaje* m)  { dest->escribe(m); } [10]
      virtual void    asignaId (ID_CANAL id=CNL_GRAL)      { identificador = id; }
    public :
                      CanalGral (ID_CANAL id=CNL_GRAL) : Proceso(0) // Proceso pasivo
                                                           { identificador = id; }
      ID_CANAL    ..  leeId ()                             { retura identificador; }
};
```

1.4. Proceso emisor/receptor que delimita el tipo de canales a utilizar

La clase *EmiRec* puede enviar y recibir mensajes por cualquier canal, ***EmisorReceptor*** es una clase que origina objetos que están limitados a emplear sólo un subgrupo de la jerarquía de canales, ésta es la de los canales clasificados de acuerdo a la forma de sincronización en la comunicación (observar las figuras 5.1 y 5.3).

Seudocódigo de la clase *EmisorReceptor*

```
clase EmisorReceptor : public EmiRec
{
  protected:
    ESTADO envia  (Mensaje* m, Canal* c)                  { return EmiRec::envia(m,c); }
    ESTADO recibe (Mensaje*& m, Canal* c)                 { return EmiRec::recibe(m,c); }
    virtual void asignaId (ID_EMI_REC id = EMI_REC_NO_CNX)  { identificador = id; }
  public:
    EmisorReceptor(ID_EMI_REC id = EMI_REC_NO_CNX) : EmiRec(id) {}
};
```

[8] Es posible otorgar amistad a las operaciones de *envia(...)* y *recibe(...)* de *EmiRec* y no a toda la clase.
[9] Por razones de eficiencia se puede sustituir por la instrucción *return fte->msjCom;*
[10] Por lo expuesto en la nota previa, se puede tener *dest->msjCom = m;*

1.5. Definición de la sincronización en la comunicación mediante canales

En el modelo previo de canal, no se abordó el modo de sincronización en la comunicación, a continuación se agrega la resolución de este aspecto. Se consideran los modos más difundidos : el paso de mensajes asíncrono y el paso de mensajes síncrono, siendo que son simples, pueden implementarse de manera razonablemente eficiente y tienen el potencial necesario para permitir al programador expresar la mayoría de los programas distribuidos.

Siendo fieles a la búsqueda de una organización que agrupe a los canales de acuerdo a ciertos patrones de clasificación se añade la clase ***Canal***, que si bien no amplía lo ya expuesto en *CanalGral*, aporta conceptualmente porque sirve como pivote de una ramificación que engloba a los canales dispuestos según el tipo de sincronización. En la Fig. 5.4 se puede apreciar la división de un *Canal* en *CanalSin* (canal síncrono) y *CanalAsin* (canal asíncrono). En lo que sigue se da el seudocódigo de la clase *Canal* :

1.5.1. Seudocódigo de la clase *Canal*

```
class Canal : public CanalGral
{
  public :
                    Canal (ID_CANAL id=CNL_NO_CNX) : CanalGral(id) {}
      virtual ESTADO acepta (EmiRec*)   = 0;
      virtual ESTADO entrega (EmiRec*) = 0;
};
```

1.5.2. Descripción de la comunicación y sincronización a través de un *canal asíncrono*

El **paso de mensajes asíncrono** es el mecanismo de sincronización más flexible; típicamente es el provisto por las rutinas de sistema en red y sistemas operativos de multicomputadoras [Andrews 91].

En los subsiguientes incisos se describen los lineamientos para la comunicación entre emisores y receptores mediante un ***canal asíncrono***.

a. Con paso de mensajes asíncrono, un canal mantiene un depósito de mensajes que han sido enviados; pero no recibidos.

b. El depósito de mensajes es conceptualmente sin límites de capacidad.

c. La primitiva de envío de un mensaje sobre un canal asíncrono, por parte de un proceso emisor es no bloqueante, de ahí la ejecución del envío de un mensaje no causa espera.

d. La primitiva de recepción es bloqueante. Ejecutar la recepción tiene el efecto de provocar la espera del proceso receptor hasta que haya por lo menos un mensaje en el depósito del canal.

e. Puesto que el envío y la recepción se realizan independientemente, un mensaje puede ser recibido un tiempo arbitrariamente posterior a cuando se envió.

1.5.2.1. Modelo OO de un *canal asíncrono*

El canal de conducta asíncrona es modelado como la clase designada ***CanalAsin*** (ver la Fig. 5.4) :

a. ***CanalAsin*** posee un almacén de mensajes nombrado *'buffer'*, como está acotado[11], realmente no implanta comunicación asíncrona pura.

b. El número de mesajes que puede captar *'buffer'* se señala cuando se declara un objeto de la clase ***CanalAsin***, o uno de sus derivados. En caso de omitir este señalamiento se adopta un valor por omisión.

c. *'buffer'* se implementa como una cola FIFO con prioridad; lo cual indica que el mensaje recientemente enviado por el emisor, es el último de la cola, y el mensaje pronto a ser entregado al receptor, es el primero, en caso de mensajes de igual prioridad; de otro modo, el orden de colocación de mensajes en la cola, responde a su nivel de prioridad, quedando al principio los de máxima y al extremo terminal, los de mínima.

d. ***CanalAsin*** es clase heredera de ***Canal*** y define las funciones ***acepta(...)*** y ***entrega(...)***, que en su ancestro eran virtuales puras.

e. Determina el modelo de comunicación asíncrona a través de las funciones ***acepta(...)*** y ***entrega(...)***. Ambas operaciones reportan el estado de éxito o error de la operación.

f. No existe restricción en el número de procesos emisores (fuentes) y receptores (destinos) que estén facultados para utilizar el canal. El canal puede estar declarado como global a todos ellos o como parte de algún objeto que apruebe el acceso de los procesos.

g. Los procesos emisores mandan mensajes al canal sin preocuparse cuales receptores serán los destinatarios, y viceversa, los procesos receptores toman los mensajes del canal sin importarles quienes fueron los emisores.

[11] La implantación del modelo lógico, en última instancia, recae en dispositivos físicos que necesariamente tienen recursos finitos.

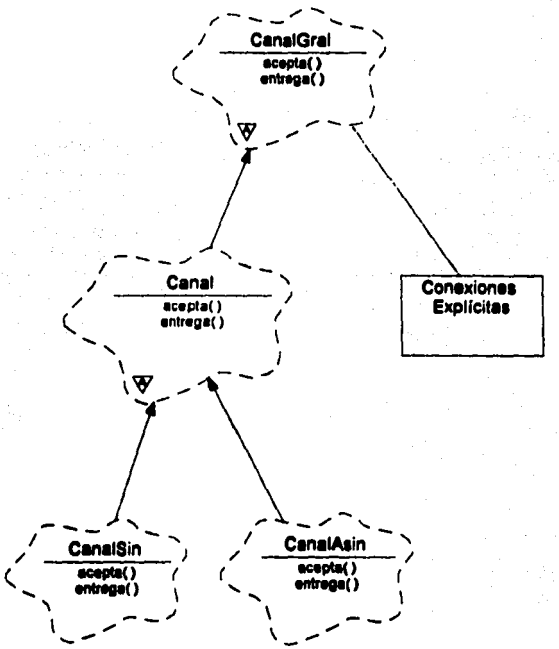

Fig. 5.4 Jerarquía de clases para canales con algún modelo de sincronización en la entrega y recepción de mensajes.

1.5.2.2. Seudocódigo de la clase *CanalAsin*

```
class CanalAsin : public Canal
{
  private:
        Cola[Mensaje*] buffer;
  public :
              CanalAsin (int tam = TAM_CNL_ASIN,ID_CANAL id = CNL_ASIN) :
                        Canal (id),buffer (tam) {}
       virtual ESTADO acepta (EmiRec*);
       virtual ESTADO entrega (EmiRec*);
};
```

La función ***acepta(EmiRec* e)***[12] recibe como parámetro el identificador del proceso que desea enviar un mensaje -en ese momento en ejecución. El mensaje se encuentra en el puerto del proceso (en el atributo *'msjCom'*). Si existe espacio en *'buffer'*, el emisor prosigue y entra a la región protegida, en ella el mensaje pasa del emisor al canal; si no, el emisor queda suspendido en espera de espacio en el *'buffer'*. El espacio se origina cuando un proceso receptor adquiere uno o más mensajes del canal, en cuyo instante la condición que detuvo al emisor desaparece pudiendo éste continuar.

```
ESTADO CanalAsin::acepta (EmiRec* e)
{
  Mensaje *m;

  select
     when (buffer.existeEspacio())
     {      m = buffer.adiciona (leeMsjCom(e));
            if (m == NULL)
               return ERROR;
            return EXITO;
     }
}
```

El contenido de la función ***entrega(EmiRec* r)***[13] es similar a la de su contraparte -la función *acepta(EmiRec* e)*-, discrepando en la condición de suspensión del proceso en ejecución. El proceso receptor es detenido cuando no existe ni un mensaje en el *'buffer'* del canal, y reanuda cuando otro proceso, algún emisor, modifica el estado del 'buffer' ejecutando una operación de envío sobre el mismo canal.

[12] El lector puede extrañarse porqué el canal no acepta y entrega directamente el mensaje en transmisión; sin embargo se juzgó conveniente pasar el identificador de los procesos, por motivos explicados en el modelo síncrono.

[13] Idem. a nota No. 11.

```
ESTADO CanalAsin::entrega(EmiRec* r)
{
   Mensaje *m;

   select
      when (buffer.existeElemento())
      {     m = buffer.extrae();
            if (m != NULL)
            {  escribeMajCom(r,m);
               return EXITO;
            }
            return ERROR;
      }
}
```

1.6.3. Descripción de la comunicación y sincronización a través de un *canal síncrono*

En el modelo de paso de mensajes asíncrono, la operación de envío es no bloqueante. Esto acarrea tres consecuencias. Primero, un proceso emisor puede estar arbitrariamente adelantado con respecto al proceso receptor. Segundo, no se tiene la certeza sobre la recepción de los mensajes, a no ser por mensajes de confirmación, lo cual en caso de fallas es crítico. Tercero, los mensajes deben ser almacenados en un depósito y en la práctica éste tiene un espacio finito. El modelo de **paso de mensajes síncrono** evita estas consecuencias porque la comunicación y la sincronización están estrechamente acopladas **[Andrews 91]**.

Las características generales para este tipo de comunicación se suscriben a continuación :

a. En particular, el envío y recepción son bloqueantes. Si un proceso trata de enviar un mensaje a un canal síncrono, espera hasta que otro proceso esté aguardando a recibir de ese mismo canal.

b. Así, el paso de un mensaje representa un punto de sincronización entre dos procesos, una cita[14] entre ellos. El canal no necesita acumular mensajes.

c. Si el emisor prosigue, entonces el mensaje fue en verdad entregado.

d. El efecto de la comunicación síncrona es una asignación distribuida, con la expresión siendo evaluada por el proceso emisor y después asignada a la variable en el proceso receptor.

[14]Conocido en la literatura concurrente como 'rendezvous'.

1.5.3.1. Modelo OO de un *canal síncrono*

Se extiende la jerarquía de canales modelados hasta ahora, con una nueva clase llamada ***CanalSin*** (observar la Fig. 5.4), cuyas particularidades se especifican en los puntos subsecuentes :

a. ***CanalSin*** es clase heredera de *Canal*. Define las operaciones de ***acepta(...)*** y ***entrega(...)*** para coordinar de forma síncrona los procesos emisores y receptores que hacen uso de sus servicios.

b. ***CanalSin*** mantiene dos colas de procesos[15], una de emisores y otra de receptores. La política aplicada por omisión es ***FIFO***, cuando la prioridad de los procesos es pareja, cuando no, el orden de atención es preferente para el de mayor prioridad, que será el que encabece la cola de procesos en espera.

c. Un emisor queda suspendido en la cola de procesos emisores, al tratar de enviar un mensaje no existiendo en el canal ningún receptor en espera.

d. Complementariamente, un proceso receptor es bloqueado en la cola de procesos receptores, si ningún emisor ha acudido a la cita para el paso del mensaje.

e. Un proceso emisor despierta al primer proceso receptor pendiente en la cola de receptores.

f. Un proceso receptor despierta al primer proceso emisor que espera en la cola de emisores.

g. No hay límite en el número de procesos usuarios del canal.

h. Los emisores pueden desconocer la identidad de los receptores, y éstos de los primeros, si el canal que comparten es declarado global. Sin embargo, la información es dirigida directamente a un proceso si es el canal es declarado local a él (aunque con permiso de acceso a los demás).

[15]Puede emplearse nada más una cola de procesos puesto que nunca hay procesos emisores y receptores paralizados al mismo tiempo.

1.5.3.2. Seudocódigo de la clase *CanalSin*

Primeramente se exhibe la declaración de la clase, con la definición de algunas funciones en línea, después la definición de las funciones restantes.

```
class CanalSin : public Canal
{
  private:
      Cola[Process*] emisores, receptores;
      Process        *emisor, *receptor;
      void    despiertaProceso(Object *);
      void    suspendeProceso(Queue &);
  public:
      void    acepta(EmisorReceptor *fuente);
      void    entrega(EmisorReceptor *destino);
      BOOL  receptoresPendientes ()    { return (receptores.size() != 0);  }
      BOOL  emisoresPendientes ()      { return (emisores.size()  != 0); }
  };
```

Las operaciones ***receptoresPendientes(...)*** y ***emisoresPendientes(...)*** - definidas en línea- son observadoras, posibilitan conocer si existen o no procesos receptores y emisores suspendidos, respectivamente.

La función miembro ***acepta(EmiRec* e)*** de *CanalSin*, invocada por un proceso emisor en su operación de envío de un mensaje por el canal, recibe la identificación (dirección) del proceso emisor y cuestiona sobre la existencia de receptores en espera, si la respuesta es afirmativa, transmite el mensaje al receptor pendiente más antiguo[16], quedando tanto el proceso emisor como el receptor en libertad de continuar su ejecución. Si la respuesta es negativa, es decir, no existen receptores en la cola de pendientes, se suspende al emisor.

```
void CanalSin::acepta (EmisorReceptor *fuente)
{
  if (receptoresPendientes() )
  {  receptor = receptores.sub();
     escribeMsjCom((EmisorReceptor*)receptor, leeMsjCom (fuente));
     despiertaProceso(receptor);
  }
  else
     suspendeProceso(emisores);
}
```

[16]Esto depende de la política de inserción y extracción de elementos fijada para la cola de procesos en espera.

La función miembro ***entrega(EmiRec* r)*** es semejante a la función explicada previamente, con la diferencia que se lleva a cabo en el ejercicio de la solicitud de un mensajes por parte del proceso receptor -cuando ejecuta la operación *recibe(...)*. El receptor (destino) recibe el mensaje del primer proceso emisor, luego éste es despertado. Si no hay algún emisor pendiente, el receptor es suspendido.

```
void CanalSin::entrega (EmisorReceptor *destino)
{
  if (emisoresPendientes() )
  {  emisor = emisores.sub();
     escribeMsjCom (destino, leeMsjCom((EmisorReceptor *)emisor));
     despiertaProceso(emisor);
  }
  else
     suspendeProceso(receptores);
}
```

suspendeProceso(Cola[Process*] procsPendientes) extrae de '*readyQueue*'[17] a '*process*'[18] y lo coloca en la cola recibida como parámetro, en un momento dado ésta es la cola de emisores o receptores pendientes, dependiendo si el proceso en actual ejecución es un emisor esperando que el canal le acepte su mensaje, o un receptor solicitando al canal le entregue el mensaje de un emisor.

```
void CanalSin::suspendeProceso(Queue& procsPendientes)
{
  process->suspend (procsPendientes,readyQueue);
}
```

despiertaProceso(Proceso* procPendiente) incluye en *'readyQueue'* al proceso que recibe como parámetro, habilitándolo para proseguir su ejecución.

```
void CanalSin::despiertaProceso(Object *procPendiente)
{
  readyQueue.add(procPendiente);
}
```

¿ Por qué los parámetros de las funciones 'acepta(...)' y 'entrega(...)' son procesos y no mensajes ?

Cuando son varios los procesos emisores pendientes, son varios los mensajes pendientes, si cada proceso no tuviera el mensaje a enviar en su puerto, el canal tendría que encolar los mensajes, lo cual contradice el modelo síncrono. Por otro lado, de

[17] *readyQueue* representa a la cola del sistema con los procesos en facultad de ejecución.
[18] *process* representa al proceso actualmente en ejecución.

todos modos el canal tendría que mantener la cola de emisores suspendidos, además de la relación de pertenencia entre los emisores y sus respectivos mensajes, ésto para que al momento de entregar un mensaje se pueda liberar de su estado de espera a su proceso emisor correspondiente.

Con el presente modelo, al encolar a los procesos, implícitamente se están encolando a los mensajes y de un modo patente, la relación de éstos con los procesos que los contienen. Acorde a lo dicho, la operacion ***acepta(...)*** toma como parámetro la dirección del proceso que va enviar el mensaje, y ***entrega(...)***, la del proceso que lo va a recibir, ocupándose el canal de hacer la transmisión de un puerto a otro, en el modelo síncrono, o de un puerto a su depósito de mensajes y de ahí al puerto del receptor, en el modelo asíncrono.

1.6. Extendiendo el modelo propuesto para soportar comunicaciones explícitas

La infraestructura necesaria para el soporte de conexiones explícitas debe construirse en el ámbito de los procesos emisores/receptores y de los canales. Con este fin, se extiende el modelo desarrollado hasta ahora.

¿ Qué se entiende por conexiones explícitas ?

Hasta ahora, un emisor o un receptor se considera conectado al canal por el sólo hecho de usarlo en la transmisión de mensajes. Si el canal es declarado global a todos los procesos, cualquier proceso que lo nombre en su operación de envío, es un emisor para ese canal, análogamente, todo proceso que nombre a ese canal para recibir un mensaje de él, se convierte en un receptor. Mediante este método, no existe ningún control sobre el número o la identidad de los procesos que recurren al canal. Esto da las ventajas de una comunicación rápida porque se evitan validaciones, sin embargo es insegura porque deja al programador la responsablidad del buen uso de los canales.

Las conexiones[19] explícitas pueden ser vistas como registros de los procesos que tienen acceso a un determinado canal, éstos deben conectarse y desconectarse a través de operaciones validadas según el tipo de canal que se trate. El tiempo requerido para la tranamisión de los mensajes se acrecenta (en ausencia de las validaciones); pero en recompensa se adquieren las ventajas de control sobre los procesos comunicantes.

[19]Decir "con conexiones" implica la capacidad de conexión tanto como de desconexión.

1.6.1. Proceso *emisor/receptor con conexiones*

Un proceso ***emisor/receptor con conexiones*** tiene las capacidades del *emisor/receptor general*, explicado al inicio de este capítulo, más capacidades adicionales.

A continuación se exponen los puntos relevantes para un proceso *emisor/receptor con conexiones* :

a. Posee operaciones para conectar, desconectar y verificar la existencia de canales de entrada y salida.

b. Puede conectarse como fuente para cero o más canales de entrada y como destino para cero o más canales de salida.

c. Antes de realizar operaciones de envío de mensajes sobre un canal, debe conectarlo como canal de salida.

d. Previo a recibir mensajes de un canal, debe conectarlo como canal de entrada.

e. La operación de envío sobre un canal de salida, falla si no existe una conexión explícita entre dicho canal y el proceso emisor/receptor, actuando como emisor (fuente).

f. La operación de recepción de un mensaje sobre un canal de entrada, falla si no existe una conexión explícita entre dicho canal y el proceso emisor/receptor, actuando como receptor (destino).

1.6.1.1. Modelo OO del proceso *emisor/receptor con conexiones*

El proceso *emisor/receptor con conexiones* es modelado como la clase ***EmisorReceptorConex*** (visible en las figuras 5.1 y 5.3) con las siguientes características :

a. Un *emisor/receptor con conexiones* es un *emisor/receptor* general, con adición de las propiedades de conexiones, por tanto, la clase *EmisorReceptorConex* es descendiente directo de la clase *EmiRec*.

b. ***EmisorReceptorConex*** puede enviar o recibir mensajes a través de cualquier canal que soporte conexiones, ésto significa que el canal será un descendiente de la clase *CanalConex*[1].

[1] La clase *CanalConex* y sus descendientes se explicarán en puntos posteriores.

c. ***EmisorReceptorConex*** posee dos colas[2] de canales, una para los canales de entrada y otra para los de salida. Cada canal, ingresa a la cola respectiva cuando el proceso establece una conexión con él.

1.6.1.2. Seudocódigo de la clase *EmisorReceptorConex*

```
class EmisorReceptorConex : public EmiRec
{   private:
    friend class NodoEmiRecCnx;        // Por fines de implementación, para poder encolar los objetos
    friend class Cola_EmiRecCnx;       // EmisorReceptorConex en instancias de CanalConex
      Cola_CnlCnx canalesEnt, canalesSal;
  protected:
    unsigned prioridad;
    virtual void vista(void)           // Exposición de la información de la clase
  public:
                  EmisorReceptorConex (ID_EMI_REC id = EMI_REC_CNX,
                                       unsigned p = PRIOR_ER_CNX) : EmiRec(id)  { prioridad = p;  }
    virtual ESTADO envia  (Mensaje*, CanalConex*);
    virtual ESTADO recibe (Mensaje*&, CanalConex*);
    ESTADO      conectaCanalEnt (CanalConex*);
    ESTADO      conectaCanalSal (CanalConex*);
    ESTADO      desconectaCanalEnt (CanalConex*);
    ESTADO      desconectaCanalSal (CanalConex*);
    ESTADO      conectaCanalEntEmi (CanalConex*,EmisorReceptorConex*);
    ESTADO      conectaCanalSalRec (CanalConex*,EmisorReceptorConex*);
    ESTADO      desconectaCanalEntEmi (CanalConex*,EmisorReceptorConex*);
    ESTADO      desconectaCanalSalRec (CanalConex*,EmisorReceptorConex*);
    BOOL        existeCanalEntConectado (void);
    BOOL        existeCanalSalConectado  (void);
    BOOL        canalEntConectado (CanalConex* c);
    BOOL        canalSalConectado  (CanalConex* c);
};
```

La operación ***envia(Mensaje* m,CanalConex* c)*** recibe como parámetros el mensaje a enviar y el canal a utilizar para la transmisión. Revisa si el canal está conectado como canal de salida, de ser así, reutiliza la operación ***envia(Mensaje* m, CanalGral* c)*** de la clase ***EmiRec*** (emisor/receptor general). No es válido cualquier canal, sólo un ***CanalConex*** o alguno de sus descendientes. Nótese que ***CanalConex*** unifica con ***CanalGral*** debido a que el primero es descendiente del segundo.

```
ESTADO EmisorReceptorConex::envia (Mensaje* m, CanalConex* c)
{
  if (canalSalConectado(c))
     return EmiRec::envia(m,c);
  return ERROR;
}
```

[2]En particular aquí se eligió la 'cola con prioridad' por sus propiedades de acceso discriminativo, aunque realmente podría ser cualquier estructura de datos que tenga una política de acceso acorde a las necesidades de la entidad modelada.

La operación de recepción ***recibe(Mensaje*,CanalConex)*** está construida de modo similar a la operación de envío, discrepando en la naturaleza del canal pasado como parámetro, en este caso se refiere a un canal conectado como canal de entrada. La operación de *EmiRec* reutilizada es *recibe(Mensaje*, Canal*)*.

```
ESTADO EmisorReceptorConex::recibe (Mensaje*& m, CanalConex* c)
{
  if (canalEntConectado(c))
    return EmiRec::recibe(m,c);
  return ERROR;
}
```

1.6.1.2.1.Operaciones simples de conexión

conectaCanalEnt(CanalConex* c) trata de conectar el canal *'c'* como canal de entrada para el proceso que invoca esta operación. Primero verifica si *'c'* ya está conectado como canal de entrada, de estarlo no hace nada y reporta estado de éxito. Si no lo está, observa si se halla conectado como canal de salida, de ser así, se tiene una situación errónea puesto que los canales son unidireccionales invalidando la posibilidad de estar conectados simultáneamente como canales de entrada y salida para el mismo proceso, al mismo tiempo. Si tampoco está conectado como canal de salida, recién el proceso actual se registra como proceso fuente de mensajes para *'c'*, invocando la operación perteneciente a canal : ***conectaDest(this)***, al ser correcta esta operación, el proceso concluye adicionando el canal a su cola de canales de entrada a la vez que entrega el valor final de estado de éxito o error.

```
ESTADO EmisorReceptorConex::conectaCanalEnt (CanalConex *c)
{

  ESTADO edo;

  if (canalesEnt.medida() > 0  &&  canalesEnt.incluye(c))
    return EXITO;                              // 'c' ya estaba conectado como canal de entrada
  if (canalesSal.medida() > 0  &&  canalesSal.incluye(c))
    return ERROR_CANAL_NO_ENT_Y_SAL;  // 'c' es un canal unidireccional
  if ((edo = c->conectaDest(this)) != EXITO)
    return edo;                                // 'c' no pudo registrar al proceso como destino
  if (canalesEnt.adiciona(c) == NULL)
    return ERROR;                              // No pudo incluir a 'c' como canal de entrada
  return EXITO;
}
```

conectaCanalSal(CanalConex* c) conecta a *'c'* como canal de salida, después de hacer consideraciones muy semejantes a las del párrafo de explicación precedente.

```
ESTADO EmisorReceptorConex::conectaCanalSal (CanalConex *c)
{

  ESTADO edo;

  if (canalesSal.medida() > 0  &&  canalesSal.incluye(c))
     return EXITO;                                  // 'c' ya estaba conectado como canal de salida
  if (canalesEnt.medida() > 0  &&  canalesEnt.incluye(c))
     return ERROR_CANAL_NO_ENT_Y_SAL;   // 'c' no puede ser de salida siendo ya de entrada
  if ((edo = c->conectaFte(this)) != EXITO)
     return edo;                                    // 'c' no pudo registrar al proceso como fuente
  if (canalesSal.adiciona(c) == NULL)
     return ERROR;                                  // No pudo incluir a 'c' como canal de salida
  return EXITO;
}
```

desconectaCanalEnt(CanalConex* c) elimina la conexión entre el proceso que invoca la operación y *'c'*, de quien se piensa es un canal de entrada. Comprobado este hecho, sustrae el canal *'c'* de la cola de canales de entrada conectados y elimina al proceso actual del registro de procesos destino de *'c'*.

```
ESTADO EmisorReceptorConex::desconectaCanalEnt (CanalConex *c)
{
  if (canalesEnt.medida() == 0 || !canalesEnt.incluye(c))
     return ERROR_CANAL_NO_CONECTADO;
  if (canalesEnt.extrae(c) == NULL)
     return ERROR;
  return (c->desconectaDest(this));
}
```

desconectaCanalSal(CanalConex* c) es similar a la desconexión del canal de entrada.

```
ESTADO EmisorReceptorConex::desconectaCanalSal (CanalConex *c)
{
  if (canalesSal.medida() == 0 || !canalesSal.incluye(c))
     return ERROR_CANAL_NO_CONECTADO;
  if (canalesSal.extrae(c) == NULL)
     return ERROR;
  return (c->desconectaFte(this));
}
```

OBSERVACION :

Para conectar dos procesos utilizando las operaciones simples de conexión, se debe invocar la operación *conectaCanalSal(...)*, desde el proceso emisor y *conectaCanalEnt(...)*, desde el proceso receptor. Al terminar la transmisión de mensajes, cada proceso tiene la responsabilidad de ejecutar, por su parte, su respectiva operación de desconexión.

1.6.1.2.2. Operaciones compuestas de conexión

conectaCanalEntEmi(CanalConex* c, EmisorReceptorConex* e) además de conectar el canal *'c'* como canal de entrada para el proceso actual, lo conecta como canal de salida para el proceso *'e'*, estableciendo explícitamente que el proceso actual será el receptor, mientras *'e'* será el emisor, utilizando como medio de transmisión a *'c'*.

```
ESTADO EmisorReceptorConex::conectaCanalEntEmi (CanalConex* c, EmisorReceptorConex* e)
{
  ESTADO edo;

  if ((edo = conectaCanalEnt(c)) != EXITO)
    return edo;
  return (e->conectaCanalSal(c));
}
```

conectaCanalSalRec(CanalConex* c, EmisorReceptorConex* r) es la operación recíproca de la anterior, ahora *'c'* es un canal de salida para el proceso actual, y de entrada para *'r'*.

```
ESTADO EmisorReceptorConex::conectaCanalSalRec (CanalConex* c, EmisorReceptorConex* r)
{
  ESTADO edo;

  if ((edo = conectaCanalSal(c)) != EXITO)
    return edo;
  return (r->conectaCanalEnt(c));
}
```

Si existen las operaciones de conexión entre un canal y los procesos fuente y destino para él, resultan necesarias las operaciones de desconexión. La operación ***desconectaCanalEntEmi(CanalConex* c, EmisorReceptor* e)*** suprime el vínculo de *'c'* como canal de entrada para el actual proceso receptor que invoca la operación, y como canal de salida para el proceso emisor *'e'*.

```
ESTADO EmisorReceptorConex::desconectaCanalEntEmi (CanalConex* c, EmisorReceptorConex* e)
{
  ESTADO edo;

  if ((edo = desconectaCanalEnt(c)) != EXITO)
    return edo;
  return (e->desconectaCanalSal(c));
}
```

desconectaCanalSalRec(CanalConex* c, EmisorReceptorConex* r) corta la conexión entre el canal 'c' y sus procesos fuente y destino, con la misma efectividad que la operación anterior y la diferencia que si la anterior actuaba a partir del proceso receptor, aquí la acción se lleva a cabo a partir del proceso emisor.

```
ESTADO EmisorReceptorConex::desconectaCanalSalRec (CanalConex* c, EmisorReceptorConex* r)
{
  ESTADO edo;

  if ((edo = desconectaCanalSal(c)) != EXITO)
    return edo;
  return (r->desconectaCanalEnt(c));
}
```

OBSERVACION :

Para comunicar, aplicando las operaciones compuestas, un proceso emisor y un proceso receptor con un canal en medio, es suficiente invocar la operación *conectaCanalSalRec(...)*, desde el emisor ó bien *conectaCanalEntEmi(...)*, desde el receptor, aunque hacerlo desde ambos no acarrea consecuencias dañinas -si se interpreta así el valor de *ESTADO*-, sería conectar algo que ya está conectado. Para cortar el vínculo de la conexión también basta llamar la operación de desconexión únicamente desde un proceso. Hacerlo por ambos lados significaría desconectar algo no conectado.

1.6.1.2.3. Operaciones inspectoras

Las siguientes operaciones son inspectoras u observadoras de las colas de canales conectados.

existeCanalEntConectado(void) y ***existeCanalSalConectado(void)*** verifican la existencia de algún canal conectado, la primera función en la cola de canales de entrada y la segunda en la cola de canales de salida.

```
BOOL EmisorReceptorConex::existeCanalEntConectado (void)
{ return canalesEnt.medida() != 0; }
```

```
BOOL EmisorReceptorConex::existeCanalSalConectado (void)
{ return canalesSal.medida() != 0; }
```

canalEntConectado(CanalConex* c) devuelve verdadero si *'c'* está conectado como canal de entrada. ***CanalSalConectado(CanalConex* c)***, hace lo mismo si *'c'* está conectado como canal de salida.

```
BOOL EmisorReceptorConex::canalEntConectado (CanalConex* c)
{ return (canalesEnt.incluye(c));    }

BOOL EmisorReceptorConex::canalSalConectado (CanalConex* c)
{ return (canalesSal.incluye(c));    }
```

1.6.2. Descripción de un *Canal con conexiones*

Un ***canal con conexiones*** tiene el comportamiento de cualquier *canal* (acepta y entrega de mensajes), más atributos y operaciones que le permiten soportar conexiones explícitas.

En lo que sigue, se señalan las características que definen a un *canal con conexiones* :

a. Un *canal con conexiones* tiene control sobre el número e identidad de los procesos emisores y receptores conectados a él :

- Gracias a la identidad, puede saber si un determinado proceso que invoca la operación *envia(Mensaje* m, CanalConex* c)* o *recibe(Mensaje* m, CanalConex* c)*[3] está autorizado. Estar autorizado significa estar registrado en el canal 'c' como fuente o destino, lo cual sólo es posible si el proceso en cuestión conectó previamente al canal como de salida o entrada, respectivamente.

- Atendiendo al número de fuentes o destinos acreditados en un canal con conexiones, se obtiene una subclasificación de canales, una organización basada en la cardinalidad de los procesos emisores y receptores tolerados por el canal :

    - Uno a uno. El canal puede conectar únicamente un emisor y un receptor.

[3]Debe recordarse que el éxito o fracaso de estas operaciones recae en las funciones de validación de conexiones con el canal, además de las de aceptación o entrega del mensaje propiamente.

- **Uno a muchos**. El canal acepta mensajes de un emisor como máximo y los entrega a uno o más receptores.
- **Muchos a uno**. El canal establece conexión entre uno o más emisores y sólo un receptor.
- **Muchos a muchos**. El canal no tiene límites en cuanto al número de emisores y de receptores.

b. Acepta y entrega mensajes de acuerdo a algún modelo de sincronización.

c. Puede aceptar mensajes sólo de procesos emisores conectados como fuentes a él.

d. Puede entregar mensajes únicamente a procesos receptores conectados como destinos a él.

Como todas las características no pueden ser ofrecidas por un sólo ente, o al menos no es conveniente por claridad y eficiencia, siendo que en la mayoría de las aplicaciones sólo se emplea algún subgrupo, los canales con conexiones son asociados siguiendo algunos patrones, en el presente trabajo los canales con conexiones se clasifican en :

- ***Canales síncronos con conexiones*** : canales 1 x 1, 1 x N, N x 1, N x N.
- ***Canales asíncronos con conexiones*** : canales 1 x 1, 1 x N, N x 1, N x N.

1.6.2.1. Modelo OO de la clase *CanalConex*

La clase ***CanalConex*** es una especialización de *CanalGral* y es el modelo OO planteado para un *canal con conexiones*. Se trata de una clase abstracta que declara una interfaz general a ser cubierta por todos sus descendientes de acuerdo a su tipo particular. Cada tipo particular surge por consideraciones de comportamiento (asíncrono o síncrono) y cardinalidad permitida en cada extremo del canal.

Las operaciones ofrecidas por *CanalConex* permiten conectar a él procesos emisores y receptores, también, desconectarlos. Complementariamente aporta operaciones observadoras de la naturaleza del canal, de su estado y de los procesos que tiene registrados como conectados. La Fig. 5.5 indica las subdivisiones del CanalConex para diferentes tipos de canales (de acuerdo al número de los procesos conectados a sus extremos).

1.6.2.2. Seudocódigo de la clase *CanalConex*

Se presenta el seudocódigo de la clase CanalConex sin mayor explicación de sus operaciones, debido a que se tomarán en cuenta con detalle en las clases descendientes donde recién adquirirán forma específica.

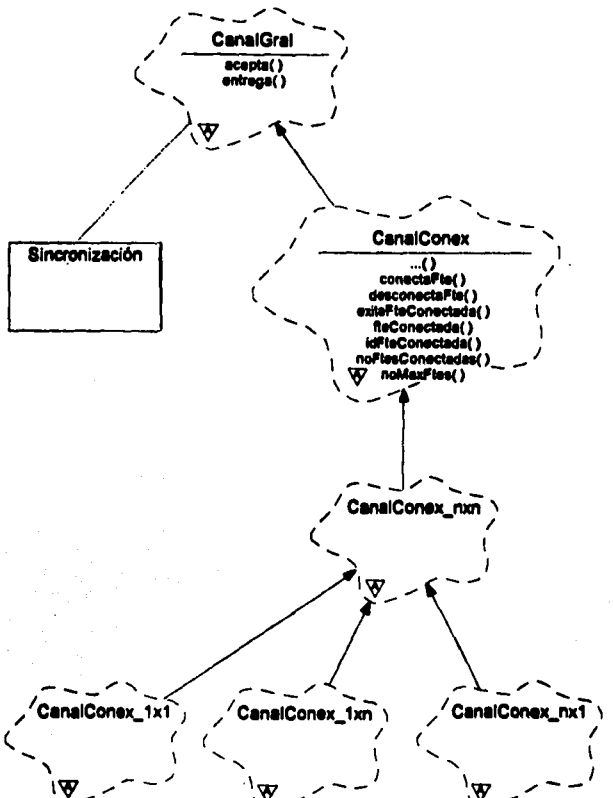

Fig. 5.5 Jerarquía de clases para canales con conexiones explícitas.

```
class CanalConex : public CanalGral
{
    friend class NodoCnlCnx;
    friend class Cola_CnlCnx;
  protected:
      unsigned prioridad;
      void vista(void)      // Exposición informativa de los datos de CanalConex
  public:
      CanalConex (ID_CANAL id = CNL_CNX) : CanalGral(id) {  prioridad = PRIOR_CNL_CNX; }
      void asignaPrior (unsigned p = PRIOR_CNL_CNX) { prioridad = p; }
      virtual ESTADO acepta (EmiRec *)                    = 0;
      virtual ESTADO entrega (EmiRec *)                   = 0;
      virtual ESTADO conectaFte (EmisorReceptorConex*)    = 0;
      virtual ESTADO conectaDest (EmisorReceptorConex*)   = 0;
      virtual ESTADO desconectaFte (EmisorReceptorConex*) = 0;
      virtual ESTADO desconectaDest (EmisorReceptorConex*) = 0;
      virtual int noMaxFtes (void)                        = 0;
      virtual int noMaxDests (void)                       = 0;
      virtual BOOL existeFteConectada (void)              = 0;
      virtual BOOL existeDestConectado (void)             = 0;
      virtual int noFtesConectadas (void)                 = 0;
      virtual int noDestsConectados (void)                = 0;
      virtual BOOL fteConectada (EmisorReceptorConex*)    = 0;
      virtual BOOL destConectado (EmisorReceptorConex*)   = 0;
      virtual EmisorReceptorConex* idFteConectada (void)  = 0;
      virtual EmisorReceptorConex* idDestConectado (void) = 0;
};
```

1.6.3. Descripción de un *canal con conexiones varios a varios* (N x N)

Un *canal con conexiones varios a varios* define atributos y operaciones que permiten controlar conexiones múltiples a sus extremos : varios emisores y varios receptores. Los atributos para almacenar el registro de dichos procesos, deben corresponder a estructuras de datos con habilidad para guardar y sustraer elementos con cierta política, además de competencia para inspeccionarlos. Las operaciones se definen con respecto a los registros almacenados.

1.6.3.1. Modelo OO de la clase *CanalConex_nxn*

El *canal con conexiones varios a varios* está representado por la clase ***CanalConex_nxn*** (ver la Fig. 5.5) descrita así :

a. Es heredera de la clase *CanalConex*, como ella, es una clase abstracta puesto que las funciones de aceptación y entrega de mensajes se mantienen como funciones puras. A diferencia suya, define las funciones para controlar las conexiones.

b. Las estructuras de datos elegidas para almacenar el registro de los procesos conectados al canal son colas FIFO con prioridad, con elementos de clase *EmisorReceptorConex*. Esta clase garantiza que no se utilizarán *emisores/receptores generales*, los cuales desconocen las conexiones, a tiempo que capacita a todas los procesos descendientes suyos (de *EmisorReceptorConex*), para usar el canal sin reparo.

Dichas estructuras de datos, tanto para los emisores conectados (*fuentes*) como para los receptores (*destinos*), deben tener la capacidad de limitar el número de elementos que puedan contener; así, aunque por omisión el *CanaConex_nxn* se crea con un número ilimitado de posibles emisores y receptores, existe la posibilidad de modificar esta cardinalidad, indicando valores diferentes como parámetros actuales en la función constructora del canal, las cantidades serán a su vez transmitidas a las funciones contructoras de las estructuras de datos de *fuentes* y *destinos* que las tomarán para restringir su cupo máximo de elementos.

1.6.3.2. Seudocódigo de la clase CanalConex_nxn

```
class CanalConex_nxn : public CanalConex
{
  private:
        Cola_EmiRecCnx fuentes, destinos;
  public:
        CanalConex_nxn (ID_CANAL id = CNL_CNX_NxN,
                        int ftesPermitidas = NO_INDEF, int destsPermitidos = NO_INDEF):
                        CanalConex(id), fuentes(ftesPermitidas), destinos(destsPermitidos) {}
        virtual ESTADO acepta (EmiRec*)     = 0;
        virtual ESTADO entrega (EmiRec*)    = 0;
        virtual ESTADO conectaFte (EmisorReceptorConex*);
        virtual ESTADO conectaDest (EmisorReceptorConex*);
        virtual ESTADO desconectaFte (EmisorReceptorConex*);
        virtual ESTADO desconectaDest (EmisorReceptorConex*);
        virtual int noMaxFtes (void);
        virtual int noMaxDests (void);
        virtual BOOL existeFteConectada (void);
        virtual BOOL existeDestConectado (void);
        virtual int noFtesConectadas (void);
        virtual int noDestsConectados (void);
        virtual BOOL fteConectada (EmisorReceptorConex*);
        virtual BOOL destConectado (EmisorReceptorConex*);
        virtual EmisorReceptorConex* idFteConectada (void);
        virtual EmisorReceptorConex* idDestConectado (void);
};
```

1.6.3.2.1.Operaciones de conexión

conectaFte(EmisorReceptorConex* fte) registra al proceso emisor *'fte'* insertándolo en la cola de procesos *fuentes*. El éxito de la operación recae en el éxito de la operación de adición de un elemento en la cola. Nótese que la función *adiciona(...)* deberá revisar si al insertar un elemento más en la estructura de datos (cola en este caso) no se sobrepasa el límite indicado como número máximo de elementos, de ser así devolverá el valor NULL, que advierte sobre el fallo en la operación.

```
ESTADO CanalConex_xxx::conectaFte (EmisorReceptorConex* fte)
{
   if (fuentes.adiciona(fte) == NULL)
      return ERR_ADICION_FTE;
   return EXITO;
}
```

conectaDest(EmisorReceptorConex* dest) registra al proceso receptor *'dest'*, adicionándolo en la cola de procesos *destinos* para el canal actual.

```
ESTADO CanalConex_xxx::conectaDest (EmisorReceptorConex *dest)
{
   if (destinos.adiciona(dest) == NULL)
      return ERR_ADICION_DST;
   return EXITO;
}
```

desconectaFte(EmisorReceptorConex* fte) revisa si el proceso 'fte' está incluido dentro de los emisores registrados en la cola de fuentes; si no está, no es necesario desconectar nada y se reporta un valor de error apropiado o incluso podría ser un valor de éxito; si está, se sustrae a 'fte' de la cola de fuentes.

```
ESTADO CanalConex_xxx::desconectaFte (EmisorReceptorConex *fte)
{
   if (!fuentes.incluye(fte))
      return ERR_FTE_NO_CONECT;
   if (fuentes.extrae(fte) == NULL)
      return ERR_ELIM_FTE;
   return EXITO;
}
```

desconectaDest(EmisorReceptorConex* dest) actúa igual que en la operación anterior, tan sólo que ahora con referencia a un proceso receptor conocido como 'dest' y la cola de destinos.

```
ESTADO CanalConex_axn::desconectaDest(EmisorReceptorConex *dest)
{
  if (!destinos.incluye(dest))
     return ERR_DST_NO_CONECT;
  if (destinos.extrae(dest) == NULL)
     return ERR_ELIM_DST;
  return EXITO;
}
```

1.6.3.2.2. Operaciones observadoras

Las operaciones de inspección de los emisores y receptores conectados al canal, dependen de las funciones observadoras de la clase cola. Enriqueciendo esta clase con funciones adicionales a las tradicionales de cola, indirectamente se estarán aportando nuevas opciones de manipulación de los registros de fuentes y destinos, ya que éstos son implementados como colas. Aquí se definen únicamente las operaciones de inspección que cubren la interfaz general declarada en su padre *CanalConex* :

```
int CanalConex_axn::noMaxFtes (void)     // Indica el número máximo de procesos fuente permitidos
{ return fuentes.medidaMax(); }          // para el canal en cuestión

int CanalConex_axn::noMaxDests (void)    // Indica el número máximo de procesos destino permitidos
{ return destinos.medidaMax(); }         // para el canal en cuestión

BOOL CanalConex_axn::existeFteConectada (void)
{ return (fuentes.medida() != 0); }          // Da verdadero si la cola de fuentes no está vacia

BOOL CanalConex_axn::existeDestConectado (void)   // Si existen elementos en la cola de destinos,
{ return (destinos.medida() != 0); }          // existe al menos un emisor conectado

int CanalConex_axn::noFtesConectadas(void)        // Cantidad de procesos fuente conectados
{ return (fuentes.medida()); }                    // actualmente al canal

int CanalConex_axn::noDestsConectados(void)       // Cantidad de procesos destino conectados
{ return (destinos.medida()); }                   // actualmente al canal

BOOL CanalConex_axn::fteConectada (EmisorReceptorConex* fte)
{ return  (fuentes.incluye(fte) != NULL);  }          // 'fte' es un proceso conectado como fuente

BOOL CanalConex_axn::destConectado (EmisorReceptorConex* dest)
{ return (destinos.incluye(dest) != NULL); }          // 'dest' es un proceso registrado como destino

EmisorReceptorConex* CanalConex_axn::idFteConectada (void)
{ return fuentes.lee();  }                    // Entrega el identificador de la primera fuente conectada

EmisorReceptorConex* CanalConex_axn::idDestConectado (void)
{ return destinos.lee(); }          // Retorna el identificador del proceso que encabeza la cola de destinos
```

1.6.4. Descripción de un *Canal con conexiones uno a uno* (1 x 1)

Un *canal con conexiones uno a uno* acepta un proceso fuente y un proceso destino, por ello define las operaciones de conexión y desconexión para autorizar esta cardinalidad. Puede ser realmente utilizado en aplicaciones cuando además de resuelto el problema de cardinalidad, define un modelo de sincronización al aceptar y entregar mensajes.

Los canales uno a uno pueden ser vistos como un subrupo de los canales N a N (varios a varios) con N = 1; es decir, pueden ser considerados similares a éstos con la restricción de una cota máxima, igual a uno, tanto para el número de procesos fuente como para los destino.

1.6.4.1. Modelo OO de la clase *CanalConex_1x1*

Un *canal con conexiones uno a uno* es conocido en la jerarquía de canales como la clase ***CanalConex_1x1*** (exaninar la Fig. 5.5), con la siguientes descripción :

- Hereda de la clase *CanalConex_nxn* las operaciones de conexión, limitando a uno el número de posibles emisores y receptores a sus extremos, lo cual lo logra construyendo la clase base con parámetros actuales iguales a la unidad, para dichos argumentos; pero no se compromete con nigún esquema de sincronización en especial, por esta razón, queda como clase abstracta dejando las operaciones de *acepta(...)* y *entrega(...)* para ser definidas posteriormente.

1.6.4.2. Seudocódigo de la clase *CanalConex_1x1*

```
class CanalConex_1x1 : public CanalConex_nxn
{
  public:
       CanalConex_1x1 (ID_CANAL id = CNL_CNX_1x1) :  CanalConex_nxn(id,1,1) {}
    virtual ESTADO acepta  (EmiRec*)     = 0;
    virtual ESTADO entrega (EmiRec*)     = 0;
};
```

1.6.5. Canales de cardinalidad mixta : *uno a varios* (1 x N) , *varios a uno* (N x 1)

1.6.5.1. Descripción de un *canal con conexiones uno a varios* (1 x N)

Un canal uno a varios es la conjunción de dos comportamientos, uno de conexión unitaria por parte del proceso fuente, y otro de conexión múltiple por parte de los procesos destino.

Modelo OO de la clase CanalConex_1xn

CanalConex_1xn es la clase que modela un *canal con conexiones uno a varios*, observable en la Fig. 5.5, que define las funciones virtuales puras de *CanalConex*, retomando las funciones de su clase padre *CanalConex_nxn*. Los valores pasados a la función constructora de su clase progenitora son la unidad y una constante tomada como número ilimitado, para los procesos emisores y receptores aceptados, respectivamente. Si se trata de violar esta cardinalidad, las funciones de conexión invocadas retornarán valores de error.

```
clase CanalConex_1xn : public CanalConex_nxn
{
  public:
      CanalConex_1xn (ID_CANAL id = CNL_CNX_1xN) : CanalConex_nxn(id,1,NO_INDEF) {}
    virtual ESTADO acepta (EmiRec*)    = 0;
    virtual ESTADO entrega (EmiRec*)   = 0;
};
```

1.6.5.2. Descripción de un *canal con conexiones varios a uno* (N x 1)

Un canal de este tipo puede puede recibir mensajes de varios emisores y entregarlos a un receptor, ésto debido a que soporta conectar uno o más fuentes, mas sólo un destino.

Modelo OO del *canal con conexiones varios a uno*

Como en el caso de *CanalConex_1xn*, la clase ***CanalConex_nx1*** (modelo OO del canal varios a uno, observable en la Fig. 5.5), resulta de la herencia de *CanalConex_nxn* con diferentes cotas máximas para *fuentes y destinos*.

```
class CanalConex_nx1 : public CanalConex_nxn
{
  public:
      CanalConex_nx1 (ID_CANAL id = CNL_CNX_Nx1) : CanalConex_nxn(id,NO_INDEF,1) {}
    virtual ESTADO acepta (EmiRec*)    = 0;
    virtual ESTADO entrega (EmiRec*)   = 0;
};
```

1.6.6. Combinando los modelos de sincronización en la comunicación con la capacidad de conexiones explícitas

Hasta ahora, todas las clases de canal que controlan las conexiones explícitas de emisores y receptores, sin importar la cardinalidad, son clases abstractas, es decir con la imposibilidad de ser empleadas para crear instancias de ellas (objetos). Esto fue exprofesamente diseñado así porque sentaron las bases de una jerarquía de canales con control de conexiones de diferente cardinalidad, independientemente de algún modelo particular de sincronización en la comunicación, para que sean las clases derivadas las que las especialicen con algún modelo.

Ahora, se busca tener canales con conexiones completos, es decir con las operaciones de control de las conexiones y con algún modelo de sincronización. Nuevamente los modelos de sincronización considerados son el síncrono y el asíncrono. Para precisar uno de estos modelos en los canales con conexiones, es suficiente definir siguiendo sus condiciones las funciones *acepta(...)* y *entrega(...)*; sin embargo si se piensa en el resto de la jerarquía de canales, se recordará que estas funciones ya fueron determinadas en las clases *CanalAsin* y *CanalSin*, lo cual sugiere que el trabajo ya esta hecho, o al menos parte de él. Y así es, se pueden reutilizar los esquemas de sincronización amalgamando las clases *CanalAsin* y *CanalSin* con las clases de canales con conexiones que controlan la cardinalidad de los emisores y receptores.

1.6.6.1. Canales asíncronos con conexiones

CanalAsinConex es una clase conceptualmente necesaria, es abstracta y sirve para agrupar a todos los canales con conexiones que tienen el modelo de sincronización asíncrono (ver la Fig. 5.6).

```
class CanalAsinConex : public CanalConex, public CanalAsin
{
  public:
     CanalAsinConex (int tambuf=TAM_CNL_ASIN, ID_CANAL id = CNL_ASIN_CNX) :
                    CanalConex(id), CanalAsin (tambuf,id) {}

  // Recibe las funciones de aceptación y entrega de mensajes de CanalAsin.
  // Las funciones para el control de conexiones se mantienen como puras.
};
```

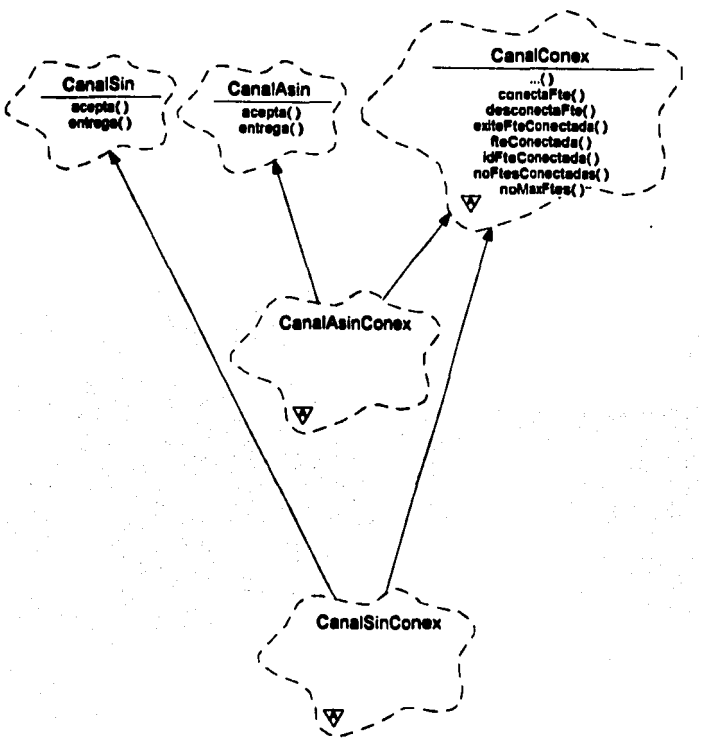

Fig. 5.6 Jerarquía de clases para canales con algún modelo de sincronización y conexiones explícitas.

1.6.6.1.1.Canal asíncrono con conexiones varios a varios

La clase ***CanalAsinConex_nxn*** combina el modelo de un canal asíncrono con el canal de conexiones múltiples en sus extremos (analizar la Fig. 5.7).

```
class CanalAsinConex_nxn : public CanalAsinConex, public CanalConex_nxn
{
  public:
    CanalAsinConex_nxn (int tambuf=TAM_CNL_ASIN, ID_CANAL id = CNL_ASIN_CNX_NxN) :
                        CanalAsinConex(tambuf,id), CanalConex_nxn(id) {}

  // Se debe pasar  a la constructora el tamaño del buffer deseado, de otro modo se adopta un valor por
  // omisión.
  // Las funciones acepta(...) y entrega(...) se definen considerando lo heredado de CanalAsinConex,
  // que viene a ser las funciones de CanalAsin.
  // Las operaciones de conexión se cubren con las operaciones heredadas de CanalConex_nxn.
};
```

1.6.6.1.2.Canal asíncrono con conexiones uno a uno

La clase ***CanalAsinConex_1x1*** acepta mensajes de un emisor y los entrega a un receptor asíncronamente (ver la Fig. 5.7).

```
class CanalAsinConex_1x1 :  public CanalAsinConex, public CanalConex_1x1
{
  public:
    CanalAsinConex_1x1 (int tambuf=TAM_CNL_ASIN, ID_CANAL id = CNL_ASIN_CNX_1x1) :
                        CanalAsinConex (tambuf,id), CanalConex_1x1(id)  {}

    // acepta(...) y entrega(...) son tomadas de la definición de la clase CanalAsinConex, la cual a su vez
    // las heredó de CanalAsin
    // Las funciones de conexión son tomadas de CanalConex_1x1, por tanto servirán para controlar que
    // hayan conectados simultáneamente sólo un emisor y un receptor
};
```

1.6.6.1.3.Canal asíncrono con conexiones uno a varios

La clase ***CanalAsinConex_1xn*** es similar a la clase *CanalConex_1x1*, en cuanto a conexiones se refiere, más las operaciones de aceptación y entrega de mensajes según el esquema asíncrono (ver la Fig. 5.7).

```
class CanalAsinConex_1xn :  public CanalAsinConex, public CanalConex_1xn
{
  public:
    CanalAsinConex_1xn (int tambuf=TAM_CNL_ASIN, ID_CANAL id = CNL_ASIN_CNX_1xN) :
                        CanalAsinConex(tambuf,id), CanalConex_1xn(id) {}

    // Cubre las funciones acepta(...) y entrega(...) con las correspondientes de CanalConexAsin.
    // Cubre las funciones de conexión con las funciones de la clase CanalConex_nx1.
};
```

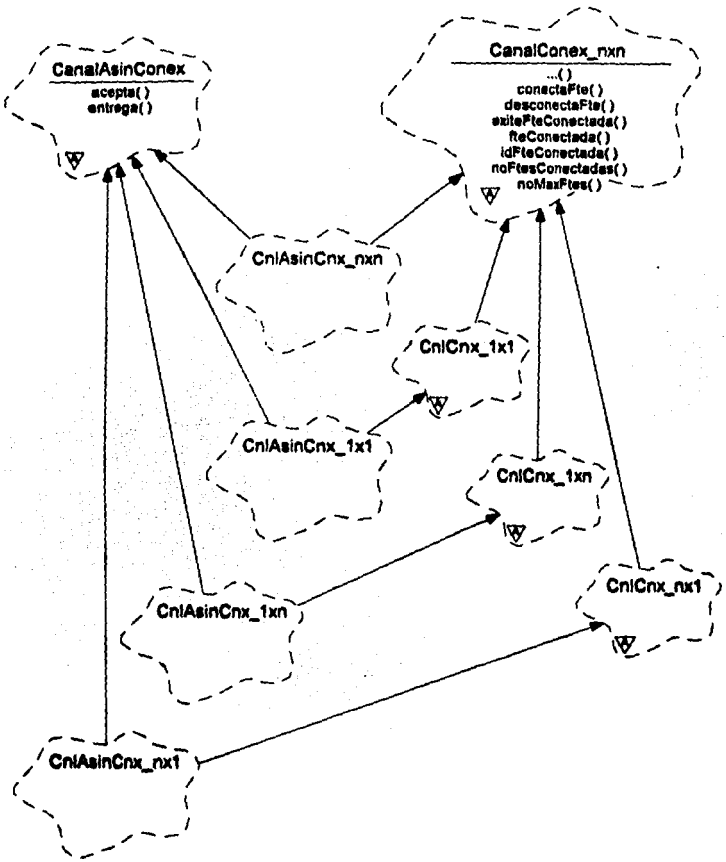

Fig. 5.7 Jerarquía de clases para canales con modelo de sincronización asíncrono y conexiones explícitas.

1.6.6.1.4.Canal asíncrono con conexiones varios a uno

La clase ***CanalAsinConex_nx1*** (ver la Fig. 5.7) conecta y desconecta varios emisores y un receptor como lo indica su clase padre ***CanalConex_nx1***, y sincroniza la aceptación y entrega de mensajes según su otro progenitor, la clase ***CanalAsinConex***.

```
class CanalAsinConex_nx1 : public CanalAsinConex, public CanalConex_nx1
{
  public:
    CanalAsinConex_nx1 (int tambuf=TAM_CNL_ASIN, ID_CANAL id = CNL_ASIN_CNX_Nx1) :
                        CanalAsinConex(tambuf,id), CanalConex_nx1(id) {}

    // Cubre las funciones acepta(...) y entrega(...) con las correspondientes de CanalConexAsin.
    // Cubre las funciones de conexión con las funciones de la clase CanalConex_nx1.
};
```

1.6.6.2. Canales síncronos con conexiones

La construcción de canales con conexiones y comportamiento síncrono es similar a la de los canales asíncronos con conexiones, solo que esta vez la clase pivote para agrupar esta jerarquía (conectada a la jerarquía de todos los canales), se denomina ***CanalSinConex*** y se forma a partir de la clase ***CanalSin*** y ***CanalConex*** (ver la Fig. 5.6).

La clase ***CanalSinConex*** se especializa en (ver la Fig. 5.8) :

Clase *CanalSinConex_1x1* : combina las funciones síncronas de aceptación y entrega de mensajes de CanalSin, recibidas a través de ***CanalSinConex***, con las funciones de conexión de la clase ***CanalConex_1x1***.

Clase *CanalSinConex_nxn* : conjunta la interacción síncrona de ***CanalSinConex*** y las conexiones de ***CanalConex_nxn***.

Clase *CanalSinConex_1xn* : es un canal síncrono por ser clase derivada, aunque indirecta de ***CanalSin***, y con control conexiones de tipo uno a varios porque también desciende de ***CanalConex_1xn***.

Clase *CanalSinConex_nx1* : hereda las operaciones de sincronización de ***CanalSinConex***, y las operaciones de conexión de ***CanalConex_nx1***.

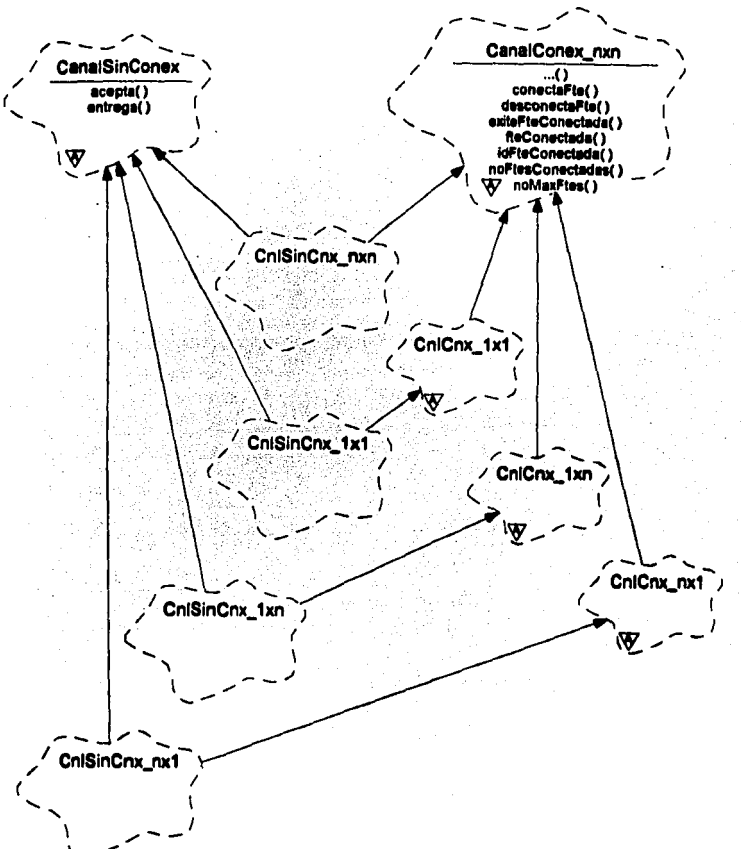

Fig. 5.8 Jerarquía de clases para canales con modelo de sincronización síncrono y conexiones explícitas.

Paralelamente a la visualización gráfica de las clases en los diagramas de jerarquías, se pueden observar los elementos modelados a través de una simbología que a continuación se expone :

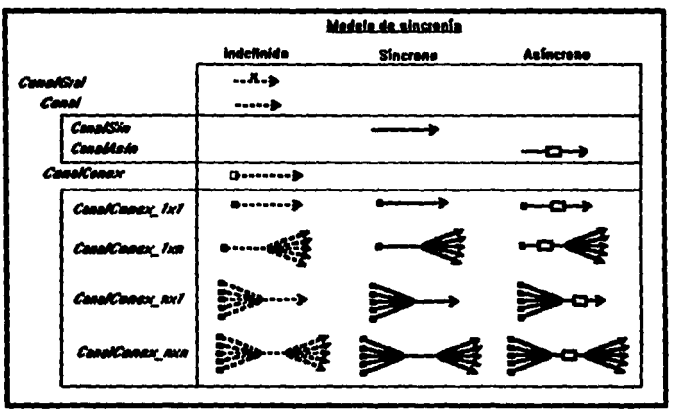

Fig. 5.9 Símbolos de *canales*.

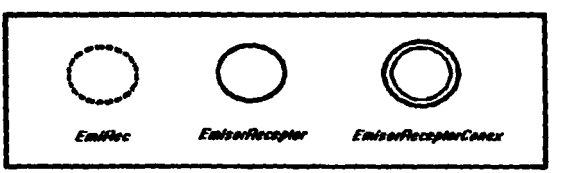

Fig. 5.10 Símbolos de los procesos *emisores/receptores*.

2. EJEMPLOS DE APLICACION

A continuación se ilustra el uso de los mecanismos desarrollados mediante ejemplos simples, pero fundamentales, dado que muestran la transferencia de mensajes desde unos procesos a otros, a través de algunos de los canales de la jerarquía modelada en este trabajo. En primer término se ejemplifica el uso de un canal sin conexiones, en segundo lugar, el de un canal con conexiones explícitas.

2.1. Envío - recepción de mensajes empleando canales sin conexiones

Los procesos (un emisor y un receptor) usados para el ejemplo son construidos a partir de la clase *EmisorReceptor*, porque si bien requieren las capacidades de envío y recepción de mensajes, no es necesario que controlen explícitamente la conexión de los canales que emplean. Estos procesos corresponden a las clases *EjEmiMsj* y *EjRecMsj* (ver la Fig. 5.11), ambas con el parámetro formal de *Canal*, lo que les permite en un momento dado trabajar con una instancia de *CanalAsin* o *CanalSin*, sin que la decisión de trabajar con una u otra implique algún cambio en la definición de las clases, ésto gracias a la jerarquía y uniformidad que siguen los canales, a pesar de variar en el modelo de sincronización.

EjEmiMsj es un **emisor de mensajes enteros**, de tantos mensajes como el número pasado a su contructora en el parámetro 'lim' :

```
class EjEmiMsj : public EmisorReceptor
{
    MensajeEntero  msj;          // Un No. entero es el contenido de los mensajes que transmite
    MensajeControl msjCtrl;      // Un mensaje de control para indicar el final de la transmisión
  public:
    EjEmiMsj(char*, char*, int, Canal*);    // Constructora donde se determina el comportamiento
};                                          // del proceso emisor ejemplificado


EjEmiMsj::EjEmiMsj (char* nomEmi, char* nomCnl, int lim, Canal* c) : EmisorReceptor(EJ_EMIMSJ)
{
    int i;
    ESTADO edoOp = EXITO;

    for (i = 0; i < lim && edoOp == EXITO; i++)
    {  msj.escribeContenido(i);
       cout << nomEmi << '(' << i+1 << ')' << " : " << "Mensaje a enviar -- "
            << nomCnl << " --> " << msj.leeContenido() << "\n";
       edoOp = envia(&msj,c);       // Envía un mensaje por algún objeto compatible con Canal
    }
    edoOp = envia(&msjCtrl,c);          // Envía la señal que indica fin de la transmisión
    if (edoOp == EXITO)                 // Revisa el estado de las operaciones de envío
       cout << nomEmi << "TERMINA EXITOSAMENTE !!!\n";
    else
       cout << nomEmi << "TERMINA SIN TOTAL DE ENVIOS !!!\n";
}
```

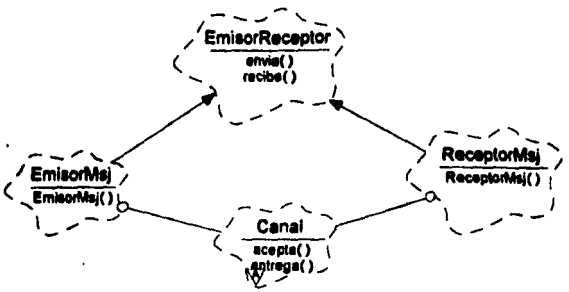

Fig. 5.11 Diagrama de clases para el ejemplo de envío y recepción de mensajes mediante un canal sin conexiones.

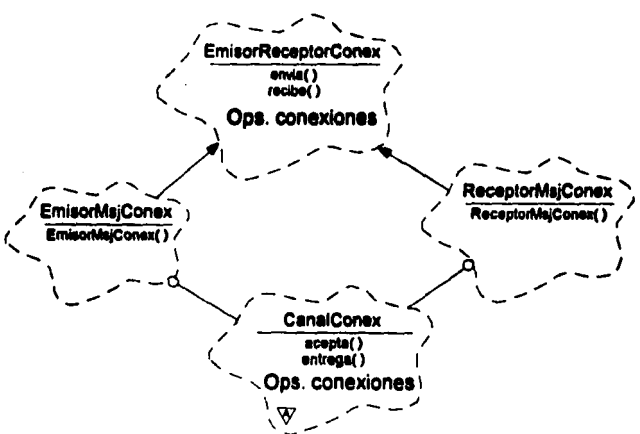

Fig. 5.12 Diagrama de clases para el ejemplo de envío y recepción de mensajes mediante un canal con conexiones.

EjRecMsj es un **receptor de mensajes enteros**, en actividad constante de recepción hasta que capta una señal que le indica finalización :

```
clase EjRecMsj: public EmisorReceptor
{
    MensajeEntero *msj;        // Mensaje en el que recibe la información de tipo entero
    Mensaje *msjAux;           // Mensaje general en el que capta cualquier tipo de mensaje
  public:
    EjRecMsj(char*, char*, Canal*);      // Constructora del proceso receptor
};

EjRecMsj::EjRecMsj (char* nomRec, char* nomCal, Canal* c) : EmisorReceptor(EJ_RECMSJ)
{
    int i = 0;
    ESTADO edoOp = EXITO;
    ID_MSJ id;

    do
    {  edoOp = recibe(msjAux,c);      // Recibe un mensaje a partir del objeto 'c' (una variedad de canal)
       id = msjAux->leeId();          // Interpreta la clase de mensaje recibido
       if (id == MSJ_CONTROL || edoOp != EXITO)  // Si es señal de fin, termina el ciclo de recepción
          break;
       msj = (MensajeEntero*)msjAux;             // Utiliza el mensaje de información recibido
       cout << "          " << msj->leeContenido() <<  " <-- " << nomCal
            << " -- Mensaje recibido" << " : " << nomRec << '(' << ++i << ')' << "\n";
       delete(msj);
    } while (TRUE);
    if (edoOp == EXITO)                  // Revisa el estado de las operaciones de recepción
       cout << "                          " << nomRec << "TERMINA EXITOSAMENTE !!!\n";
    else
       cout << "                          " << nomRec << "TERMINA SIN TOTAL RECEPCION !!!\n";
}
```

El diagrama de clases del programa puede observarse en la Fig. 5.11, mientras que en la Fig. 5.13 se tiene su gráfica simbólica. El seudocódigo siguiente corresponde a la declaración de los objetos, siendo el canal en este caso, un objeto de *CanalSin* (canal síncrono).

```
int main(void)
{  clrscr();
   cout<< "          EMISION - RECEPCION DE MENSAJES\n";
   cout<< "      Mecanismo : CANAL SINCRONO SIN CONEXIONES\n";

   CanalSin  c1;     // Declaración de canal síncrono. Si se desea la transmisión con política asíncrona,
sólo                 // tendría que alterarse esta línea del programa, además del título obviamente
   EjEmiMsj   a("Emisor A ",'A-B', 5,&c1);          // Emisor de mensajes por el canal 'c1'
   EjRecMsj   b("Receptor B ",'A-B',&c1);           // Receptor de mensajes por el canal 'c1'

   Null  dos;
   return (0);
}
```

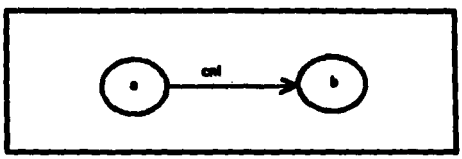

Fig. 5.13 Ejemplo de envío-recepción de mensajes por un canal síncrono sin conexiones explícitas.

2.2. Envío - recepción de mensajes empleando canales con conexiones

Este ejemplo es muy parecido al incluido previamente, con la variante que ahora la interacción entre los procesos se realiza empleando alguna derivación de un canal con conexiones, lo cual determina que las clases para el emisor y receptor ejemplificados deban contar con las operaciones de conexión explícita. El parámetro formal correspondiente al canal utilizado para la comunicación está definido como de la clase *CanalConex*, permitiendo a los procesos trabajar con cualquier cardinalidad de conexión y modelo de sincronización (ver la Fig. 5.12).

```
class EmisorMajConex : public EmisorReceptorConex
{    MensajeEntero msj;
  public:
     EmisorMajConex (char*, char*, int, CanalConex*);
};

EmisorMajConex::EmisorMajConex (char* nomEmi, char* nomCnl, int lim, CanalConex *c) :
                                   EmisorReceptorConex(EJ_EMIMSJ_CNX)
{ int i;   ESTADO edoOp = EXITO, edoCnx;

  if ((edoCnx = conectaCanalSal (c)) != EXITO)      // Conecta a 'c' como canal de salida
     cout << nomEmi << " : ERROR CONEXION " << edoCnx << ": CNL TIPO " << c->leeId() << "\n";
  else
  { for (i = 0; i < lim && edoOp == EXITO; i++)
     { delay (RETARDO_EMISION);                     // Prepara la información a enviar
       msj.escribeContenido(i);
       cout << nomEmi << '(' << i+1 << ')' << " : " << "Mensaje a enviar -- "
            << nomCnl << " --> " << msj.leeContenido() << "\n";
       edoOp = envia(&msj,c);                       // Envía el mensaje por el canal 'c'
     }
     if (edoOp == EXITO)
       cout << nomEmi << " TERMINA ENVIOS CON EXITO !!!\n";
     else
       cout << nomEmi << " TERMINA SIN TOTAL DE ENVIOS !!!\n";
     if (desconectaCanalSal (c) != EXITO)           // Desconecta el canal 'c'
       cout << "          " << nomEmi << " : ERROR DESCONEXION " << edoCnx << "\n";
  }
}
```

```
class ReceptorMajConex : EmisorReceptorConex
{    Mensaje *majAux;
     MensajeEntero *maj;
  public:
     ReceptorMajConex (char*, char*, int, CanalConex*);
};


ReceptorMajConex::ReceptorMajConex (char* nomRec, char* nomCnl, int lim, CanalConex *c) :
                                        EmisorReceptorConex(EJ_RECMSJ_CNX)
{  int i;    ESTADO edoOp = EXITO, edoCnx;

   if ((edoCnx = conectaCanalEnt (c)) != EXITO)          // Conecta a 'c' como canal de entrada
      cout << "          " << nomRec << " : ERROR EN CONEXION "
           << edoCnx << ": CNL TIPO " << c->leeId() << "\n";
   else
   {  for (i = 0; i < lim && edoOp == EXITO; i++)
      {   edoOp = recibe(majAux,c);                      // Recibe un mensaje por el canal 'c'
          if (edoOp == EXITO)
          {  maj = (MensajeEntero*)majAux;
             cout << "          " << maj->leeContenido() << " <-- " << nomCnl << " -- "
                  << " Mensaje recibido" << " : " << nomRec << '(' << i+1 << ')' << "\n";
             delete(maj);
             delay (RETARDO_RECEPCION);                  // Proceso la información recibida
          }
      }
      if (edoOp == EXITO)
         cout << "          " << nomRec << " TERMINA RECEPCIONES CON EXITO !!!\n";
      else
         cout << "          " << nomRec << " TERMINA SIN TOTAL RECEPCION !!!\n";
      if (desconectaCanalEnt (c) != EXITO)             // Desconecta el canal 'c'
         cout << "          " << nomRec << " : ERROR EN DESCONEXION " << edoCnx << "\n";
   }
}
```

A continuación se presentan dos ejemplos diferentes de la función principal de un programa concurrente, como se puede apreciar, para variar el canal utilizado por la aplicación no requiere más cambio que el de la declaración (ver las figuras 5.14 y 5.15).

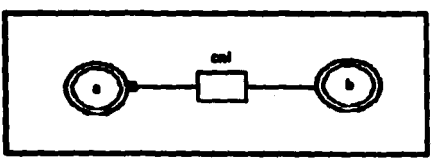

Fig. 5.14 *Ejemplo de envío-recepción de mensajes mediante un canal asíncrono con control explícito de conexiones 1 a 1.*

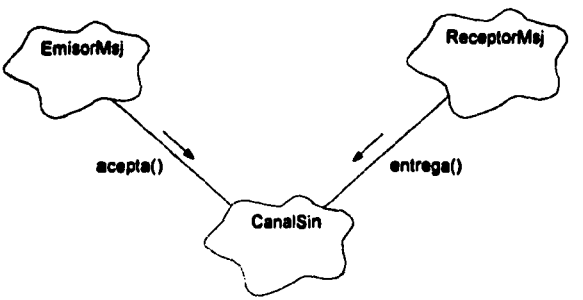

Fig. 5.13 Diagrama de objetos para el ejemplo de envío y recepción de mensajes mediante un canal síncrono sin conexiones explícitas.

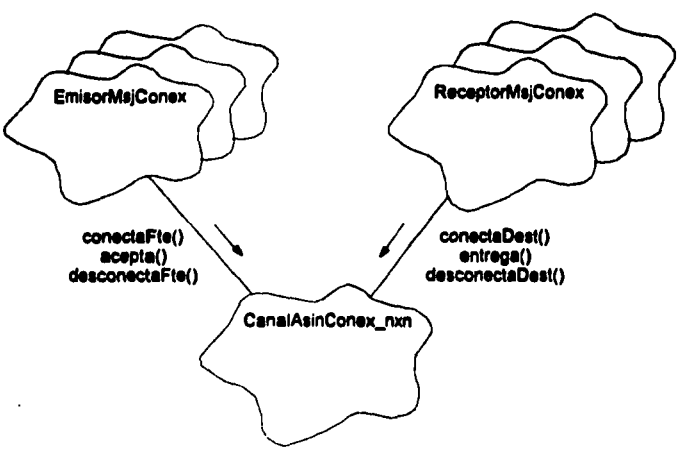

Fig. 5.14 Diagrama de objetos para el ejemplo de envío y recepción de mensajes mediante un canal asíncrono con conexiones 'N' a 'N'.

Ejemplo 1 :

```
int main ()
{ clrscr();
  cout<< "          EMISION - RECEPCION DE MENSAJES\n";
  cout<< "    Mecanismo : CANAL ASINCRONO (1 a 1) CON CONEXIONES\n\n";

  CanalAsinConex_1x1 cal(15);          // Declaración de un canal asíncrono con conexiones 1x1

  EmisorMsjConex    emi("Emisor A "  ,"A 1x1 B",9,&cal);
  ReceptorMsjConex  rec("Receptor B ","A 1x1 B",5,&cal);

  Null os;
  return (0);
}
```

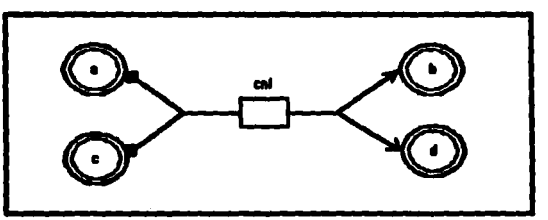

Fig. 5.15 Ejemplo de envío-recepción de mensajes mediante un canal asíncrono con control explícito de conexiones N a N.

Ejemplo 2 :

```
int main ()
{ clrscr();
  cout<< "          EMISION - RECEPCION DE MENSAJES\n";
  cout<< "    Mecanismo : CANAL ASINCRONO (N a N) CON CONEXIONES\n\n";

  CanalAsinConex_nxn cal(15);        // Declaración de un canal asíncrono con conexiones Nx N

  EmisorMsjConex    emiA("Emisor A "  ,"AC nxn BD",4,&cal);
  EmisorMsjConex    emiC("Emisor C "  ,"AC nxn BD",4,&cal);

  ReceptorMsjConex  recB("Receptor B ","AC nxn BD",3,&cal);
  ReceptorMsjConex  recD("Receptor D ","AC nxn BD",5,&cal);

  Null os;
  return (0);
}
```

CAPITULO 6

JERARQUIAS DE CLASES DE MECANISMOS DE INTERACCION DE ALTO NIVEL

Son varias y diferentes las formas como los grupos de procesos pueden interactuar en un programa distribuido. En este capítulo se presentan algunos paradigmas de interacción, sincronización y comunicación, que dan como resultado mecanismos llamados de alto nivel, porque asumen para su construcción los mecanismos básicos (canales y emisores/receptores de mensajes), desarrollados con anterioridad.

La clave para entender los programas basados en paso de mensajes, consiste en comprender las restricciones y libertades que cada proceso tiene para comunicarse con los demás, éstas son ***condiciones de sincronización y comunicación que marcan la forma de interacción entre los procesos.***

Cada paradigma de interacción, es un ejemplo o modelo de un patrón de comunicación y una técnica de programación asociada, que puede ser utilizada para resolver una colección de interesantes problemas de programación distribuida [Andrews 91].

Dentro de cada paradigma de interacción, se pueden distinguir procesos con características definidas, los cuales se pueden modelar y ofrecer como clases base, para que llegado el momento se ajusten a las necesidades propias de cada problema de aplicación. Con la metodología OO, es posible llevar a cabo esta adaptación de forma natural y sencilla, tratando al máximo de no repetir esfuerzo en análisis, diseño e implementación, si las clases base son pensadas como mecanismos estandar[1]. Algunos paradigmas son suceptibles a ser abstraidos y traducidos en mecanismos que sin perder su generalidad, son aplicables a varios problemas y sí le ahorran al programador bastante trabajo; sin embargo hay otros, que pueden tener tantas variantes, que no posiblitan factorizar su comportamiento en un modelo general y a la vez útil para luego ser especializado en clases derivadas.

1. ¿ Cómo se especializan los mecanismos de interacción ?

Las clases base de los mecanismos ofrecidos tienen secciones que son fijas para el paradigma de interacción al que pertenecen, independientemente de la aplicación; pero otras están sometidas a ella. Estas porciones variables, intercaladas entre las fijas, marcan un comportamiento especial del proceso y son fragmentos cuya definición se difiere hasta las clases herederas. Para ilustrar este esquema, que es muy frecuente y similar para varios mecanismos y procesos en este capítulo, se presenta el seudocódigo del *MecanismoX*, un mecanismo imaginario, representativo de cualquiera que tenga partes fijas referidas a acciones de preparación y culminación, más una parte variable con acciones especiales establecidas por un descendiente.

El seudocódigo es el siguiente :

```
class MecanismoX : public EmisorReceptor
{ private:
                // Miembros útiles para el MecanismoX.
    introduccion (...)
            { /* Acciones fijas de introducción del MecanismoX. */ }
    conclusion (...)
            { /* Acciones fijas de conclusión del MecanismoX. */ }
```

[1]Mecanismo estandar, en el sentido que son los más frecuentes en aplicaciones.

```
  protected:
                // Miembros comunes para el MecanismoX y sus herederos.

      virtual comportamientoEspecial (...)
            { /* Acciones determinadas por los herederos del MecanismoX; sin embargo,
                 puede ser conveniente definir un comportamiento considerado por omisión . */ }
  public:
      MecanismoX (...)  // Definición del comportamiento del MecanismoX.
            { introduccion (...);
              comportamientoEspecial (...);
              conclusion (...);
            }
}


class MecanismoX_enProblemaY : MecanismoX
{
    comportamientoEspecial (...)
          { /* Acciones acordes a las necesidades del problema de aplicación que se trate. */ }
}


main ()
{           // Utilización del MecanismoX adecuado a un problema específico : el problema Y.

    ProcesoTipo1_ProblemaY     usuarioA, usuarioB;
    MecanismoX_enProblemaY     mecanismo (...);
    ProcesoTipo2_ProblemaY     usuarioC, usuarioD;
}
```

Nótese que la construcción de las clases correspondientes a los procesos usuarios, también puede ser resultado de derivaciones semejantes a las del mecanismo, ésto dependerá del paradigma de interacción al que pertenezcan, de ser así, juntos forman un conjunto preestablecido de componentes que puede ser reutilizado, con poco o ningún trabajo adicional, el de las especializaciones.

En este esquema se espera que al declarar el *MecanismoX_enProblemaY*, se ejecute el constructor del *MecanismoX*, con la invocación de la última definición de la función virtual, es decir la perteneciente al *MecanismoX_enProblemaY*. Lamentablemente, el lenguaje de implementación utilizado no acepta este diseño.

En C++Concurrente [Olmedo 91], el comportamiento del proceso se define en la constructora de la clase, siendo que la constructora, por propiedad heredada del C++, invoca las funciones definidas en su ambiente, de tal suerte que al ejecutarse la construcción del *MecanismoX*, se invoca la función *comportamientoEspecial(...)* de su clase y no la de su heredero. Este problema se solucionaría si C++Concurrente no definiera el comportamiento de los procesos en la constructora sino en cualquier otra función[2].

[2]Actualmente, el autor del lenguaje estudia esta posibilidad.

Haciendo uso del lenguaje como está ahora, se opta por pasar al mecanismo un objeto que encierra el comportamiento instanciado a un problema dado. La forma como se pasa este objeto, determina si el mecanismo se especializa estática o dinámicamente.

1.1. Especialización estática

En este modo de especialización, el objeto que encierra los atributos y funciones instanciadas a un problema particular, se pasa como parámetro a la constructora del mecanismo general, es decir :

```
class MecanismoX : public EmisorReceptor
{ private:
        introduccion (...)  { ... }
        conclusion (...)    { ... }
  public:
        MecanismoX (..., MecX_enProblemaA especializacion)
                { introduccion (...);
                  especializacion.comportamientoEspecial (...);
                  conclusion (...);
                }
}


class MecX_enProblema
{
        virtual comportamientoEspecial (...) = 0;
                // Puede ser útil definir por 'default', el comportamiento más probable de ser
                  // recurrido por varias aplicaciones.
}


class MecX_enProblemaY : MecX_enProblema
{
        comportamientoEspecial (...)
                { Acciones acordes a las necesidades al problema de aplicación que se trate. }
}
```

Es conveniente observar que en este modelo, la función de *comportamiento especial()* no tiene acceso directo a los elementos del *MecanismoX*. Si se desea obtenerlo, se deberá recurrir a declaraciones de amistad y/o al recurso de paso de parámetros.

```
main ()
{       // Utilización del MecanismoX adecuado a un problema específico : el problema Y.

    ProcesoTipo1_ProblemaY      usuarioA, usuarioB;
    MecX_enProblemaY              mecX_especial;
    MecanismoX                    mecanismo (...,mecX_especial);
    ProcesoTipo2_ProblemaY      usuarioC, usuarioD;
}
```

1.2. Especialización dinámica

En este modelo existen mensajes que llevan el objeto que especializa a un mecanismo dado, llamemos a estos mensajes de clase MsjEspecializador; dichos mensajes son enviados por uno o más procesos ocupados en especializar el mecanismo (a tiempo de ejecución); el mecanismo, por su parte, para cada mensaje recibido sensa si se trata de una especialización y de ser diferente a la actual, utiliza el mensaje para actualizar su parte variable, que bien puede ser por ejemplo una condición de filtraje o difusión[3]. Descrito a manera de seudocódigo se tiene lo siguiente :

```
class MecanismoX : public EmisorReceptor
{ private:
      MsjEspecializador* especializacion;
      Mensaje* msj;
      introduccion (...)  { ... }
      conclusion (...)    { ... }
  public:
      MecanismoX (..., CanalGral* canal)
      { introduccion (...);
          recibe(canal, msj);
        if (msj->esTipoEspecializacion())          // TRUE, si msj corresponde a una especializacion.
        {  if (msj->revisaVersion(mEspec))         // TRUE, si se debe actualizar el mecanismo
             especializacion = msj->leeContenido();          // con una nueva especializacion.
        }
        else     // msj es un mensaje normal de información a ser procesada por el mecanismo
             especializacion->recupera_comportamientoEspecial (...);  // Trabaja la información de
        conclusion (...);                                              // acuerdo a su especialización.
      }
}


class MsjEspecializador : Mensaje
{
      // Atributos y funciones de cualquier mensaje.
      // Atributos y funciones para especialización de algún mecanismo entre ellas, por ejemplo :
        comportamientoEspecial (...) { ... }
}


class Especializador : EmisorReceptor
{ private:
      MsjEspecializador especializacion;
  public:
      Especializador (CanalGral* canal)
      {     // Prepara el Mensaje de especialización especializacion.
            envia (canal, &especializacion);     // Especializa al mecanismo conectado a 'canal'.
            // Prepara mensajes de datos
            // Envia los mensajes de datos al mecanismo recien especializado.
       }
}
```

[3] Estas condiciones se explicarán posteriormente al describir los mecanismos *filtro* y *difusor*.

```
main ()
{       // Utilización del MecanismoX adecuado a un problema específico : el problema Y.

    CanalAsin     canal;

    Especilizador   usuario;
    MecanismoX    mecanismo (...,canal);
          // Se sobreentiende que pueden haber otros procesos usuarios conectados al canal,
          // su rol, dependerá del problema de aplicación que se trate.
}
```

Como ejemplo de la forma que se propone para la modelización con la metodología OO, de mecanismos de sincronización y comunicación más complejos que los del capítulo previo, en este capítulo se consideran los mecanismos citados a continuación :

- ***Puente.***
- ***'Pipe'.***
- ***Buffer*** para comunicación síncrona.
- ***Difusor*** de mensajes.
- ***Sumidero*** de mensajes.

Tomando en cuenta estos mecanismos, se implantan los siguientes paradigmas de interacción :

- Flujo por *puentes.*
- Flujo por *pipes.*
- Interacción *productor - consumidor.*
- Interacción *productor - buffer - consumidor.*
- Difusión de mensajes vía un *difusor.*
- Absorción de mensaje vía un *sumidero.*
- Propagación de mensajes a través de *propagadores.*

Se adoptó la forma de especialización estática para evitar introducir detalles que oscurezcan el objetivo de cada diseño.

Todos los modelos están acompañados por ejemplos que los ilustran, siendo en general, ejemplos de la aplicabilidad de las clases de *canales* y *emisores/receptores* de *mensajes* del capítulo 5, independientemente que los desarrollos de este capítulo sean tomados como nuevos mecanismos.

2. Descripción del mecanismo de interacción *Filtro*

Un ***filtro*** es un ente discriminador que cuenta con ciertos patrones de calificación o análisis, en base a los cuales retiene, consume o deja pasar, intactos o transformados, los elementos que le llegan. En el mundo de la comunicación entre procesos, un ***filtro*** es un ***proceso con un flujo de información de entrada y otro de salida, siendo la salida función de la entrada***. Por tanto, la especificación correcta de un proceso ***filtro*** es aquella en la que los valores de los mensajes enviados por sus canales de salida, están relacionados con los valores de los mensajes recibidos por sus canales de entrada.

La utilidad del proceso ***filtro*** para resolver problemas, alcanza su grado más alto cuando está conectado a varios ***filtros***, formando con ellos una red de procesos que actúan de forma paralela. Un ***filtro*** en una red comparte sus canales con los procesos vecinos, la única restricción es la compatibilidad de canales e información que para un ***filtro*** son de salida y para sus vecinos son de entrada.

Dentro de la categoría de procesos ***filtro*** se distinguen dos variantes : ***sin memoria*** (puente) y ***con memoria*** (pipe).

2.1. Descripción de un *filtro sin memoria* : *puente*

Topográficamente, existen lugares que permanecerían aislados de no ser por la construcción de un ***puente*** que los una. Análogamente, en una red de procesadores no siempre existen enlaces directos de cada uno ellos con los restantes; sin embargo, por imposiciones del problema, los procesos alojados en procesadores no vecinos pueden requerir la comunicación entre sí, ignorando las restricciones físicas. Por ejemplo, si un proceso ***A***, ubicado en la máquina ***1***, desea comunicarse con un proceso ***B***, ubicado en la máquina ***N***, siendo que no hay una conexión física directa entre las máquinas ***1*** y ***N***, los mensajes enviados por ***A***, antes de llegar a ***B***, deben cruzar tantos intermediarios como máquinas formen el camino comprendido entre las máquinas ***1*** y ***N***. Estos intermediarios son ***puentes***, ***puentes*** de software que sirven como repetidores de la información.

Por lo dicho, un ***puente*** es un mecanismo con un canal de entrada y uno de salida, por los que ***recibe y transmite mensajes sirviendo de nexo entre los procesos conectados a él***, los procesos pueden ser del mismo tipo ***puente***. Para resolver el problema del ejemplo citado, conocido como de comunicación remota, basta con ubicar un ***puente*** en cada máquina que forma parte del camino físico que une a las máquinas ***1*** y ***N***; los ***puentes*** deben estar conectados entre sí.

Un *puente*, como tal, no realiza ningún tratamiento de los mensajes recibidos; sin embargo, en el presente trabajo se le desea ***otorgar cierta 'inteligencia' permitiéndole, opcionalmente, transformar la información antes de reenviarla o absorberla.*** La operación de filtraje es sencilla y no necesita de almacenamiento de datos, razón por la cual se clasifica a *puente* como un ***filtro sin memoria.***

2.1.1. Modelo OO de *puente*

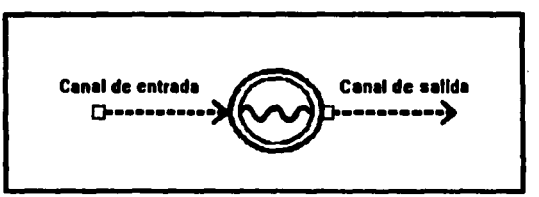

Fig. 6.1 Símbolo del mecanismo *Puente*.

Un *puente* en el modelo de objetos es conocido como la clase ***Puente;*** su símbolo se aprecia en la Fig. 6.1, su diagrama de clases en la Fig. 6.2 y su descripción teórica en los siguientes puntos :

a. Un ***Puente*** es un objeto con vida propia que hasta concluir (cuando recibe un mensaje de control para finalizar), recibe, filtra (si es necesario) y envia mensajes, por ello es derivación del proceso con estas capacidades : el *EmisorReceptor.*

b. Está conectado a dos canales, uno de entrada y otro de salida, ambos de clase ***CanalConex***, lo que permite a ***Puente*** trabajar con cualquier modelo de sincronización y cardinalidad.

c. Ofrece la posibilidad de transformar y absorber la información a través de una función de filtraje. Al tratar de otorgar libertad de especializar esta función, sin modificar a ***Puente***, se tropieza con el problema explicado en el punto (1), solucionándolo como ahí se trató :

> en lugar de ajustar la función de filtraje derivando la clase ***Puente***, se deriva la clase ***FiltroPuente*** que es pasada como parámetro al constructor de ***Puente***, esa clase puede ser vista como un dispositivo de filtraje que puede ser especializado.

d. La información que pasa por ***Puente*** puede ser de cualquier tipo, con la condición que sea modelada como descendiente de la clase ***Mensaje***.

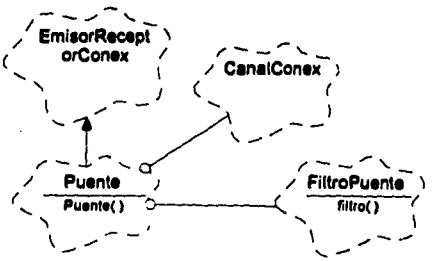

Fig. 6.2 Jerarquía de clases del mecanismo 'Puente'.

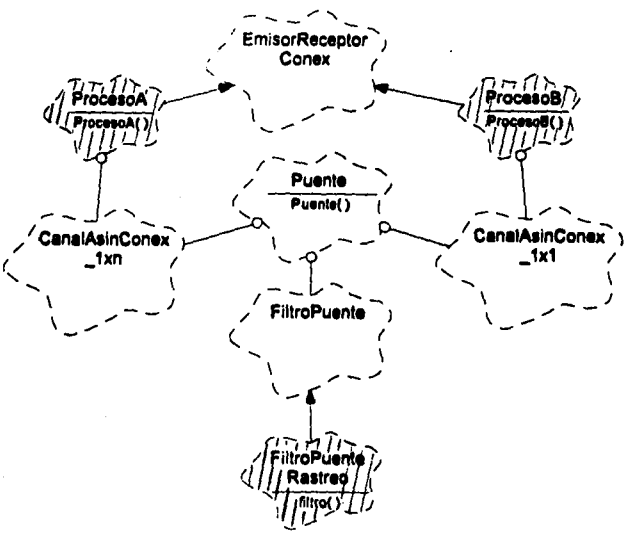

Fig. 6.3 Jerarquía de clases del ejemplo de aplicación de 'Puente' : "Conexión remota con rastreo de mensajes".

2.1.2. Seudocódigo de la clase *Puente* y sus clases asociadas

```
class Puente : public EmisorReceptorConex
{
        Mensaje *informacion;
  public:
        Puente (CanalConex*, CanalConex*, FiltroPuente&);
};
```

La conducta de un Puente se desarrolla en su función constructora :

```
Puente::Puente (CanalConex* entrada, CanalConex* salida, FiltroPuente& fil)
{
     conectaCanalEntrada (entrada);
     conectaCanalSalida (salida);
     do
     { recibe (informacion, entrada);
        if (fil.filtro (información))
           envia (información, salida);
     } while (informacion->leeId() <> MSJ_CTRLFIN)
     desconectaCanalEntrada (entrada);
     desconectaCanalSalida (salida);
}
```

La clase *FiltroPuente* es la clase general de filtraje. La función *filtro(...)* se plantea con el siguiente protocolo a ser cumplido por las clases derivadas :

- puede modificar el contenido de mensaje,
- devuelve TRUE si el mensaje debe ser transmitido por el canal de salida, FALSE, si es absorbido, es decir si no necesita ser reenviado.

```
class FiltroPuente
{ public:
    virtual BOOL filtro (Mensaje*)  { return TRUE;  }
                // Por omisión todos los mensajes que recibe, los reenvía intactos.
};
```

2.1.3. Ejemplo de aplicación del mecanismo *puente* : "Conexión remota con el rastreo del camino seguido por cada mensaje para llegar a su destino".

Se retoma el problema de conexión remota, con la característica adicional del cálculo estadístico del tiempo requerido por cada mensaje para llegar a su destino. Los procesos que intervienen cumplen la siguiente especificación :

• *Proceso A* emite contínuamente mensajes de prueba, iniciandolos con el identificador de la máquina en la que reside y la hora en que los envía.

• *Proceso B* recibe los mensajes y saca estadísticas basándose en el contenido de los mismos.

• Los *puentes* radican en diferentes máquinas o procesadores.

• Cada *puente* añade al contenido del mensaje los datos de la máquina a la que pertenece y la hora actual, indicando con ésto la máquina que atraviesa el mensaje y el momento en que lo hace.

• *Proceso A* está conectado a varios puentes, los cuales compiten por ser los intermediarios en el paso de los mensajes.

• El tiempo transcurrido desde *proceso A* a *proceso B* será varible, dependiendo del camino que siga el mensaje y del grado de ocupación de las máquinas que atraviesa.

• El contenido de cada mensaje guarda el registro de identificadores de las máquinas que cruza, además del momento en que lo hace.

2.1.3.1. Modelo OO del ejemplo de aplicación de *puente*

La Fig. 6.3 exhibe el diagrama de clases[4] del ejemplo de aplicación, la Fig. 6.4, el diagrama de objetos y la siguiente figura expresa gráficamente el problema :

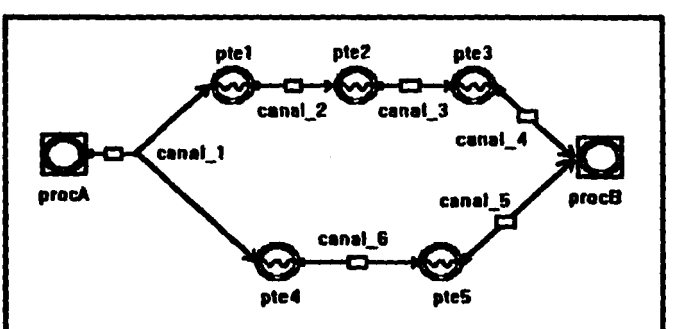

Fig. 6.5 Ejemplo de aplicación de Puente : "Conexión remota con rastreo de mensajes".

[4] Las clases con líneas sesgadas son las únicas que debe definir el usuario, las demás son clases predefinidas.

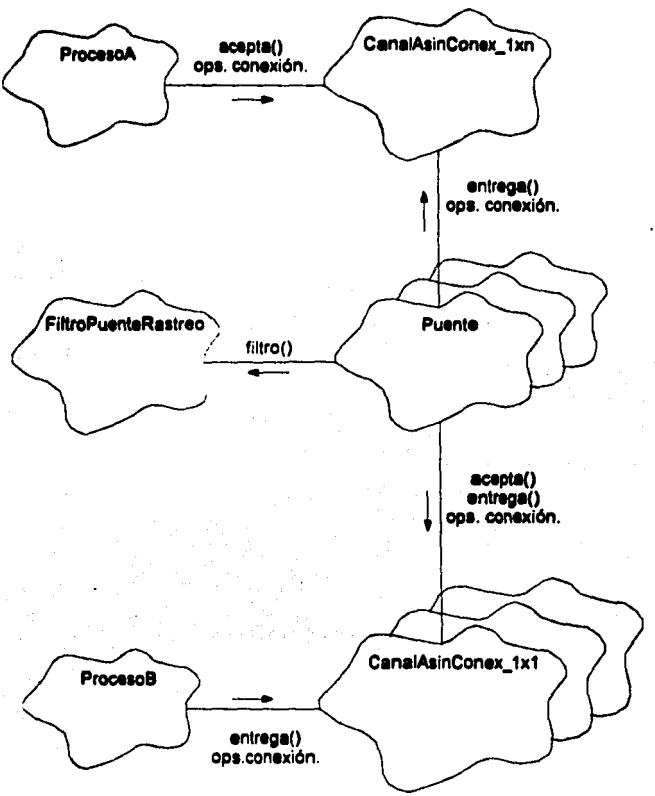

Fig. 6.4 Diagrama de objetos del ejemplo de aplicación de 'Puente' : "Conexión remota con rastreo de mensajes".

Cabe destacar lo siguiente :

• Los dos procesos extremos *A* y *B* se mapean a las clases *ProcesoA* y *ProcesoB*, herederas de la clase *EmisorReceptorConex*.

• *Puente 1*, *Puente 2*, ... son instancias de la clase *Puente* conectados a diferentes canales.

• *ProcesoA* está conectado a *CanalAsinConex_1xn* como fuente y los *Puentes 1* y *2* como destinos. Por la naturaleza del canal uno a muchos, el destino más veloz o menos ocupado será el que acuda primero por cada mensaje.

• Los *Puentes 1* y *2*, *2* y *3* entre sí, así como los *Puentes 4* y *5*, están conectado mediante canales de tipo *CanalAsinConex_1x1*.

• Los *Puentes 3* y *5* son fuentes para el *CanalAsinConex_nx1* cuyo destino es el *ProcesoB*, habilitado por este canal para recibir mensajes de cualquiera de las fuentes indistintamente.

• Cada mensaje para este problema, conocido como de clase *MsjRastreo*, es derivada de la clase *Mensaje*, con un contenido definido como una serie de registros donde cada uno denota el identificador de una máquina y un campo tipo hora.

2.1.3.2. Seudocódigo de la solución al problema de aplicación de *Puente*

El tipo de los mensajes en circulación obedece los siguientes lineamientos :

```
class Registro
{   Tiempo   horaDePaso;
    IdMaq    maqDePaso;
        // Funciones pertinentes para su manipulación.
};

class MsjRastreo : Mensaje
{   Cola[Registro]  contenido;
        // Declaración y definición de las funciones restantes  para un mensaje de este tipo.
};
```

La reutilización de Puente en esta aplicación sólo implica definir :

```
class FiltroPuenteRastreo : FiltroPuente
{   BOOL filtro (Mensaje* m)
    {   Registro *reg;
            // Crea un nuevo reg con los datos de la máquina y hora actuales.
            // Adiciona reg al contenido de m.
    }
};
```

A continuación el seudocódigo de los procesos distantes en comunicación :

```
class ProcesoA : EmisorReceptorConex
{       MsjRastreo  mRastreo;
  public:
        ProcesoA (CanalAsinConex_1xn* canalSal)  // Definición de su comportamiento.
        {     conectaCanalSalida(canalSal);
              while (TRUE)
              {     // Inicia el contenido de mRastreo con el identificador de la máquina
                    // y la hora presentes.
                    envia (mRastreo, canalSal);
              }
        }
};


class ProcesoB : EmisorReceptorConex
{       MsjRastreo *mRastreo;
          // Otros datos necesarios para obtener las estadísticas.
  public:
        ProcesoB (CanalAsinConex_1x1 canalEnt)
        {     conectaCanalEntrada (canalEnt);
              while (TRUE)
              { recibe (mRastreo,canalEnt);
                    // Obtiene el contenido del mensaje y lo interpreta para mantener una tabla
                    // de datos de máquinas y tiempos ocupados para el envío de los mensajes.
              }
        }
};
```

Solución del problema de aplicación de *Puente*

```
main ()                          // Todos los procesos en actividad concurrente.
{                                // La capacidad de los canales es el valor por omisión.
  CanalAsinConex_1xn  canal_1;
  CanalAsinConex_1x1  canal_2, canal_3, canal_5;
  CanalAsinConex_nx1  canal_4;

  ProcesoA            procA (&canal_1);
  FiltroPuenteRastreo fil;
  Puente              pte1(&canal_1,&canal_2,fil), pte2 (&canal_1,&canal_3,fil),
                      pte3(&canal_2,&canal_4,fil), pte4 (&canal_3,&canal_4,fil),
                      pte5 (&canal_4,&canal_5,fil);
  ProcesoB            procB (&canal_5);
}
```

2.2. Descripción de un *filtro con memoria* : *pipe*

Un *pipe* es un proceso *filtro* con dos únicos canales para conducir la información en filtración : un canal de entrada, para recibir los datos a examinar, y un canal de salida, para retransmitir los datos que considera necesitan un análisis mayor. La información que considera suficientemente filtrada, la retiene como parte de la solución total.

Cuando se conecta la salida de un *pipe* como la entrada de otro, repitiéndose tal situación cuantas veces el problema de aplicación lo requiera, se dice que los procesos establecen una conección en red tipo *pipeline*. En el presente modelo se construye el pipeline en forma dinámica, es decir, no existe un número fijo de pipes, éstos se crean según las necesidades de filtraje de cada aplicación. Se tomó esta elección porque :

- se libera al programador de que sea él quien lance los procesos en paralelo, cuidando de establecer las conexiones debidas,
- existen problemas en los que la información de entrada es absorbida, de modo que al principio no se puede conocer el número de procesos necesarios, y aun de saberlo no tiene caso que todos los pipes inicien su ejecución simultáneamnete, cuando la realidad indica que son activados en cascada, a medida que les llega el primer dato a filtrar.

Los datos a filtrar, encapsulados en mensajes, se envían uno a uno al primer *pipe*, éste ejecuta su operación de filtraje y de ser necesario crea un nuevo *pipe* para que continúe la discriminación de la información. Al final, el resultado, conformado por la serie de datos filtrados, se encuentra distribuido en todos los *pipes*, cada uno de ellos debe ser capaz de entregar su fracción de resultado y si es el caso, colaborar para que la parte de los otros *pipes* también sea recolectada.

2.2.1. ¿ Cómo se recolectan los resultados en una aplicación ?

Los problemas de aplicación a resolver vía *pipeline*, tienen un origen y un final, el primero se identifica con un proceso generador de los datos de entrada y el segundo, con un proceso receptor de los resultados. Entre ambos existen uno o más *pipes* :

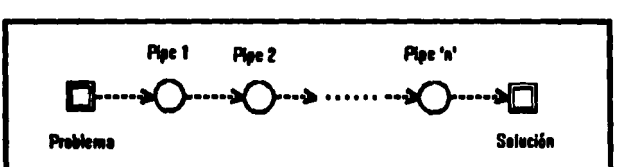

Fig. 6.6 Cadena de *Pipes* con *Problema* y *Solución* en extremos.

La información a filtrar viaja del *generador* a los *pipes*, a través de los canales nombrados con literales. Cuando concluye la operación de filtraje de todos los datos, los resultados (almacenados en la memoria local de cada pipe), pasan por los mismos canales hasta llegar a la *solución*. Los canales designados numéricamente sirven para que cada *pipe* envíe un mensaje de control a su predecesor indicándole que concluyó; el proceso predecesor, que es su creador, puede entonces tomar medidas pertinentes de destrucción del proceso que concluyó.

Como la creación del *pipeline* es dinámica, en este modelo se tienen los siguientes inconvenientes :

- Cada pipe debe detectar si proceso que debe crear a continuación debe ser solución u otro *pipe*.
- Los datos obtenidos como resultado serían locales a solución, dificultando la comunicación de esta información con otros procesos, puesto que éstos desconocen la identidad de la instancia de *solución* creada en tiempo de ejecución.
- *Solución* empezaría a recibir los resultados sólo cuando el último *pipe* hubiera terminado su operación de filtraje[5], ésto retarda la utilización de los mismos.

Para salvar estos detalles, en el modelo presentado en este trabajo, los *pipes* entregan los resultados por los canales designados numéricamente, luego, es el propio *generador* el encargado de recibir los resultados :

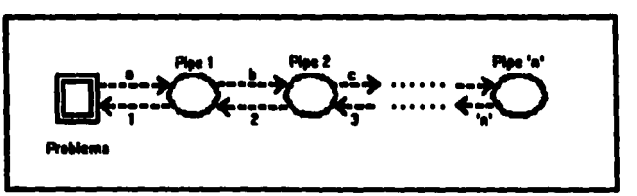

Fig. 6.7 Envío y recepción de resultados por parte de *Problema*.

A continuación se presenta una visión OO del modelo de *pipe*, más adelante se retoma la utilización de pipe ubicándolo en un esquema genaral de un problema aplicación.

[5] Si se contara con un comando alternativo tipo ALT de OCCAM, podría modelarse la recepción de datos filtrar en paralelo con la recepción y reenvío de resultados.

2.2.2. Modelo OO de filtro *pipe*

La clase ***Pipe*** es el modelo OO para caracterizar a un filtro ***pipe***, su descripción es la siguiente (ver la Fig. 6.11) :

*a. **Pipe*** es un proceso emisor y receptor de mensajes nato, su trabajo de filtraje gira en torno a los mensajes que recibe y envía. Por ello, es lógico declarar a la clase *Pipe* como heredera de ***EmisorReceptor***.

*b. **Pipe*** incluye un espacio de memoria donde almacena la información que le toca retener como parte del resultado total, información que debe entregar al finalizar el análisis de todos los datos.

c. Como en el caso de ***Puente***, ***Pipe*** ofrece la posibilidad de filtrar los datos que recibe. En este caso, la función de filtraje tiene capacidad de construir condiciones de examinación más complejas porque pueden implicar la información de la memoria del *Pipe*. También puede absorber los datos recibidos. Para guardar uniformidad, cualquier función de filtraje debe cumplir las siguientes características de interfaz :

> Recibir la información de la memoria del *Pipe*, recibir la información en proceso de filtraje y entregar la información filtrada. Las condiciones discriminativas del filtro pueden causar la modificación de la memoria y también la absorción de datos, en cuyo caso, la información en proceso de filtraje ya no debe ser retransmitida a los siguientes *Pipes*, ésto será indicado por un determinado valor de retorno de la función de filtraje (Ejm. TRUE); si la información pasa el filtro; pero debe continuar siendo examinada por otros *Pipes*, otro debe ser el valor devuelto (Ejem. FALSE).

Simbólicamente :

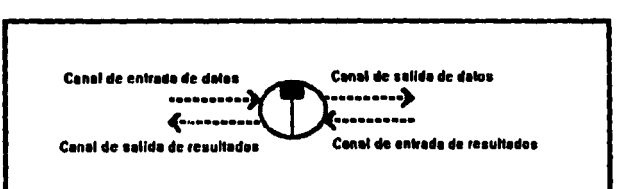

Fig. 6.8 Símbolo del mecanismo *Pipe*.

d. Cuando se crea un *Pipe*, es construido ya conectado a su creador por dos canales, uno de entrada de datos para recibir de él la información a filtrar y otro por el que le va a entregar los resultados finales. En la implementación actual, ambos canales se pasan como parámetros a la función constructora de *Pipe* :

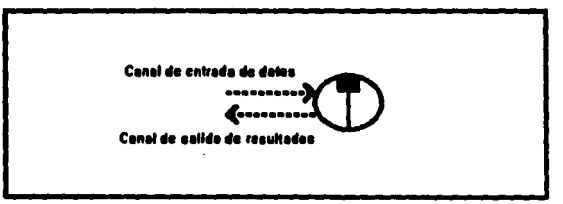

Fig. 6.9 Canales que conectan a *Pipe* con su creador.

e. Un proceso *Pipe* puede ser creador de un semejante, de darse la situación, el canal de entrada de datos del *Pipe* recién creado es el canal de salida de datos del creador, así mismo, el canal de salida de resultados finales del nuevo proceso es el canal de entrada de resultados del *Pipe* creador, es decir :

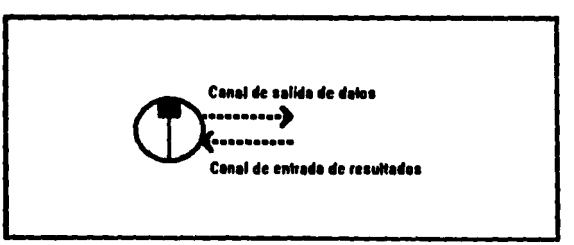

Fig. 6.10 Canales que conectan a *Pipe* con otro creado por él.

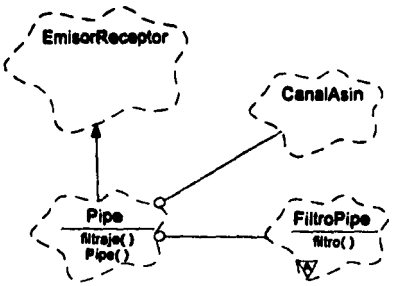

Fig. 6.11 Diagrama de clases del mecanismo 'Pipe'.

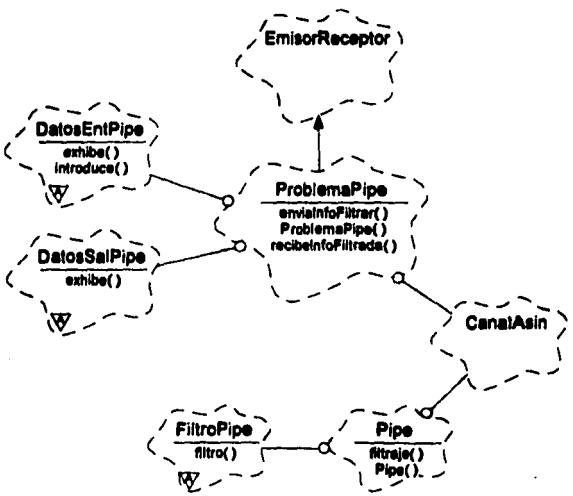

Fig. 6.12 Diagrama de clases del paradigma de interacción 'flujo entre pipes'.

1. Seudocódigo del mecanismo *Pipe*

```
class Pipe : public EmisorReceptor
{ private:
        CanalAsin  salidaDatos, entradaResultados;
                      // Canales que conectan al siguiente Pipe, dado que tienen sentido sólo
                   // cuando éste existe, podrían ser creados al mismo tiempo que él.
        Pipe       *pip;       // Siguiente Pipe.
        Mensaje    *propio, *actual, *res;
                   // propio = información de lo que se llamó memoria del Pipe.
                   // actual = información a ser filtrada por el Pipe.
                   // res    = información parcialmente filtrada, debe continuar su paso por el pipeline.
  public:
            Pipe (CanalAsin*, CanalAsin*, FiltroPipe&);      // Comportamiento del mecanismo.
        void filtraje (CanalAsin*, FiltroPipe&);
        void pasaSolucionPipesSgtes (CanalAsin*);
        void inicializaSgtePipe (void)  { pip = NULL; }
        BOOL existePipeSgte (void)    { pip != NULL; }
};
```

El comportamiento del proceso Pipe obedece a las instrucciones de su función constructora :

```
Pipe::Pipe (CanalAsin* entradaDatos, CanalAsin* salidaResultados, FiltroPipe& fil)
{ inicializaSgtePipe ();      // Al inicio nunca hay un siguiente Pipe, si es necesario, después será creado.
  recibe (propio,entradaDatos);      // Carga su memoria.
  filtraje (entradaDatos,fil);          // Filtra los datos a recibir.
  envia (propio,salidaResultados);   // Entrega su parte de resultado.
  if (existePipeSgte())              // Entrega el resultado de los Pipes siguientes.
  {  pasaSolucionPipesSgtes(salidaResultados);
     delete(pip);                    // Destruye el Pipe siguiente.
  }                                  // actual queda con el mensaje indicador de fin.
  envia(actual,salidaResultados);   // Pipe termina su actividad, así se lo señala al proceso que lo creó.
}


void Pipe::filtraje (CanalAsin* entradaDatos, FiltroPipe& fil)
{ do
  { recibe (actual,entradaDatos);          // Recibe la información a filtrar.
    if (actual->leeId () != MSJ_CTRLFIN) // Para proseguir actual no debe ser el mensaje de finalización.
     { if (fil.filtro (propio,actual,res))      // Filtra la información recibida. Devuelve TRUE si la
                                          // información debe ser filtrada por más Pipes.
       {   if (!existePipeSgte ())
               pip = new Pipe (salidaDatos,entradaResultados,fil);   // Crea el Pipe siguiente.
           envia (res,salidaDatos);          // Envía al sgte. Pipe la información filtrada parcialmente.
           }
     }
     else
        if   (existePipeSgte())             // Si no es el ultimo Pipe en el pipeline,envía al sgte. Pipe
           envia (actual,salidaDatos);   // el mensaje que indica la no existencia de más datos a filtrar.
  } while (actual->leeId () != MSJ_CTRLFIN);
}
```

```
void Pipe::pasaSolucionPipesSgtes (CanalAsin* salidaResultados)
{
  recibe (actual,entradaResultados);          // Recibe los mensajes (resultados) que le pasa el Pipe
  while (actual->leeId () != MSJ_CTRLFIN)  // siguiente, para retransmitirlos al proceso que le antecede.
  {     envia (actual,salidaResultados);      // Repite la acción hasta que recibe el mensaje
        recibe (actual,entradaResultados);   // de finalización.
  }
}
```

La clase auxiliar que modela el dispositivo de filtraje en *Pipe* es la clase *FiltroPipe*. Ella define la interfaz que debe cumplir la función virtual *filtro(...)*. Cada aplicación debe cubrir esta función, de acuerdo a sus requerimientos particulares, en alguna clase descendiente.

```
class FiltroPipe
{  public:
     virtual BOOL filtro (Mensaje*,Mensaje*,Mensaje*) = 0;
};
```

2. Paradigma de interacción : flujo de información entre *pipes*

En este apartado se plantea la infraestructura esencial para resolver cualquier problema a través de una serie de pipes, con el objetivo que el usuario del paradigma tenga únicamente que definir :

- la clase específica de mensaje que encapsula a cada dato (ejm. MsjEntero),
- los datos de entrada (o la forma de obtenerlos y mostrarlos),
- forma de exhibición de los resultados y
- la función de filtraje, encapsulada en una clase de filtraje.

Para obtener resultados a un problema dado, el usuario sólo tiene que declarar un proceso predefinido que representa al problema, instanciándolo con la clase de filtraje específica.

2.1. Modelo OO del paradigma flujo entre *pipes*

El proceso identificado como problema (de clase *Problema*) es el semillero del *pipeline* : crea al primer *Pipe*, éste al segundo, el segundo al tercero y así sucesivamente cuantos requiera la aplicación, determinada primordialmente por la función de filtraje (ver el diagrama de clases en la Fig. 6.12).

La clase *Problema* obtiene una colección de datos de entrada con los que alimenta a la serie de *Pipes*. Cuando termina la entrega de estos mensajes, recopila los mensajes ya filtrados y los devuelve como una colección de datos de salida.

Los datos de entrada y salida generalmente son numerosos, por consiguiente es favorable tenerlos organizados en una estructura que facilite su manipulación, por ejm. una *Cola*, como se eligió para el modelo actual. Ambas colecciones de datos se definen como herederas de Cola, incorporando nuevas funciones :

+ para los datos de entrada : el *llenado* de los datos en la Cola, por ejm. desde un archivo, desde el teclado, etc. Y la *exhibición* del contenido de la Cola, con enunciados alusivos al problema y con dirección a algún dispositivo de salida en especial.

+ para los datos de salida : la *exhibición* de datos, ahora referida a los datos obtenidos como resultados. El llenado de la Cola con los resultados, la realiza automáticamente Problema.

2.2. Seudocódigo de las clases que soportan el flujo entre *Pipes*

```
class ProblemaPipe : public EmisorReceptor
{
        Mensaje *m;
        MensajeControl mctrl;
        CanalAsin salDat, entRes;
        Pipe *pip;
  public:
        ProblemaPipe (DatosEntPrimo&, DatosSalPrimo&, FiltroPipe&);
        void enviaInfoParaFiltrar(DatosEntPipe&);
        void recibeInfoFiltrada(DatosSalPipe&);
};
```

El comportamiento del proceso Problema :

```
ProblemaPipe::ProblemaPipe(DatosEntPrimo& infoEnt, DatosSalPrimo& infoSal, FiltroPipe& fil)
{  int i;

   if (infoEnt.existeElem())                  // Si existe información para filtrar prosigue :
   {   pip = new Pipe(&salDat,&entRes,fil);   // Crea el primer Pipe de la cadena  de filtros.
       enviaInfoParaFiltrar(infoEnt);         // Entrega la información a filtrar.
       recibeInfoFiltrada(infoSal);           // Recolecta los resultados.
       delete(pip);                           // Elimina el Pipe creado.
    }
}
```

Envía los mensajes que alimentan al *pipeline* :

```
void ProblemaPipe::enviaInfoParaFiltrar(DatosEntPipe& infoEnt)
{
   while (infoEnt.existeElem())
   {  infoEnt.extrae(m);
      envia(m,salDat);
   }
   envia(mCtrl,salDat);
}
```

Recibe los mensajes filtrados y los almacena en la colección de datos de salida :

```
void ProblemaPipe::recibeInfoFiltrada(DatosSalPipe& infoSal)
{
   do
   {  recibe(m,entRes);
      if (m->leeId() != MSJ_CTRLFIN)
         infoSal.adiciona(m);
   } while (m->leeId() != MSJ_CTRLFIN);
}
```

La interfaz establecida para la colección de los datos de entrada :

```
class DatosEntPipe : public Cola[Mensaje*] // Se trata de una Cola -o cualquier secuencia ordenada de
{                  // mensajes-, el lenguaje utilizado realmente no permite clases genéricas; pero por
                   // fines de claridad se utiliza esta nomenclatura en la presentación del modelo.
  public:
      DatosEntPipe(int capacidad) : ColaMsj(capacidad) {}
      virtual void introduce(void)    = 0;
      virtual void exhibe(void)       = 0;
};
```

Modelo para la serie de datos de salida :

```
class DatosSalPipe : public Cola[Mensaje*]
{  public:
      DatosSalPipe(int capacidad) : ColaMsj(capacidad) {}
      virtual void exhibe(void)  = 0;
};
```

2.3. Problema de aplicación : reconocedor de números primos

Dada una colección de números (entre 2 y un límite dado por el usuario), se quiere conocer cuáles son primos. Para solucionar el problema se utiliza una versión paralela de la Criva de Eratóstenes, el algoritmo dice que un número es primo si no es divisible por ninguno de los números primos menores a él.

2.3.1. Modelo OO de la solución en base a las clases del flujo entre Pipes

Los resultados son los números primos, por tanto, cada uno de ellos debe quedar en la memoria de cada *Pipe*, para que luego sean transmitidos al recolector de resultados (el objeto de clase *Problema*). Al recibir cada dato a examinar, el filtro de cada *Pipe* revisa si es múltiplo del número primo que almacena, si lo es lo absorbe, de otro modo autoriza que sea enviado al siguiente *Pipe* (ver el diagrama de clases en la Fig. 6.13 -recuérdese que sólo las clases con líneas sesgadas corresponden a las clases que el usuario debe definir- y el diagrama de objetos en la Fig. 6.14).

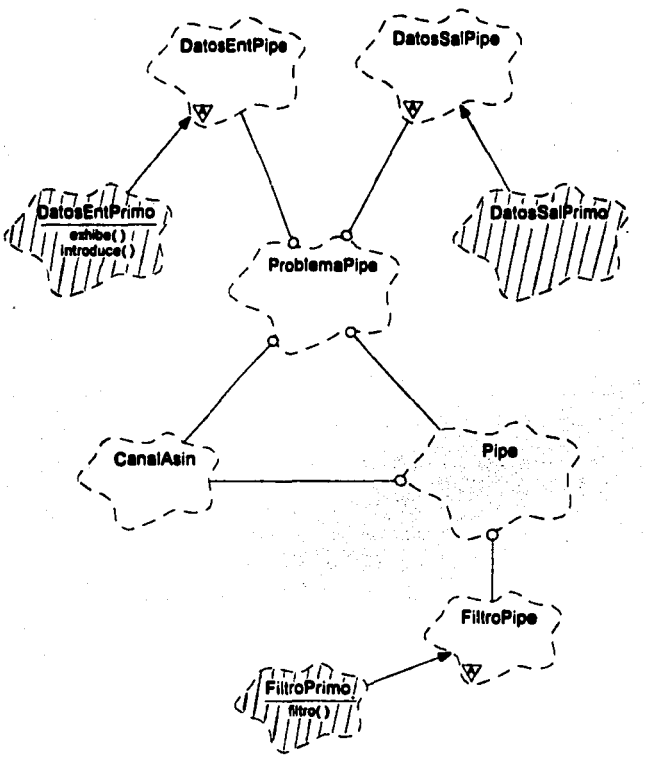

Fig. 6.13 Diagrama de clases del ejemplo de aplicación del paradigma de interacción 'flujo entre pipes' : " Reconocedor de números primos".

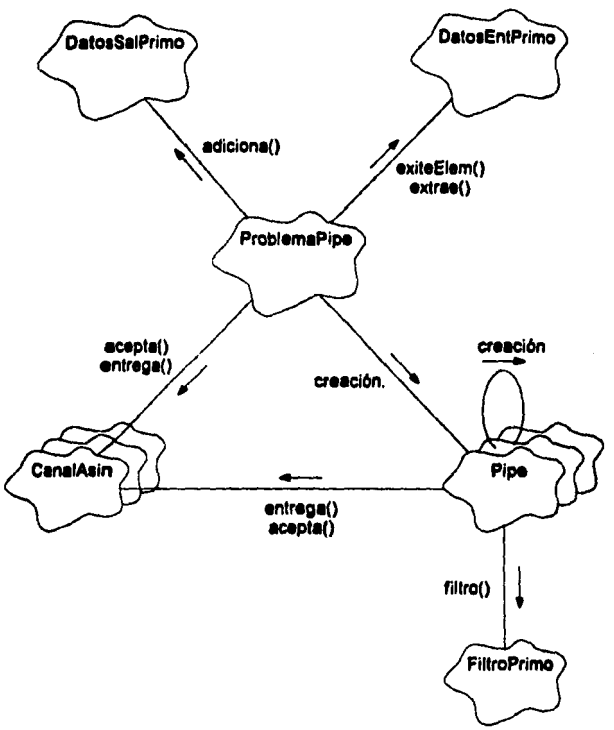

Fig. 6.14 **Diagrama de objetos del ejemplo de aplicación del paradigma de interacción 'flujo entre pipes' : "Reconocedor de números primos".**

2.3.2. Seudocódigo de la solución

La clase *FiltroPrimo* es la que reglamente la aplicación presente :

```
class FiltroPrimo : public FiltroPipe
{  public:
      BOOL filtro (Mensaje*,Mensaje*,Mensaje*);
};


BOOL FiltroPrimo::filtro (Mensaje* propio,Mensaje* enPaso,Mensaje* result)
   // Retorna TRUE si existe algun dato va a ser transmitido al prox pipe
{
   if ((enPaso->leeContenido() % propio->leeContenido()) != 0)
   {  result = enPaso;          // Si el dato a filtrar no es múltiplo del almacenado:
      return TRUE;                 // El dato a ser transmitido es el mismo enPaso.
   }                               // Autoriza la transmisión al siguiente Pipe.
   else                            // Otro :
   {  delete(result);              // Absorbe el mensaje recibido enPaso.
      return FALSE;
   }
}
```

Definición de la clase de los datos de entrada :

```
class DatosEntPrimo : public DatosEntPipe
{
      int cantDatos;
   public:
         DatosEntPrimo (int capacidad) : DatosEntPipe(capacidad) { cantDatos = 0; }
      void introduce (void);
      void exhibe (void);
};
```

Introducción y exposición de los números a filtrar :

```
void DatosEntPrimo::introduce (void)
{  int i;
   MensajeEntero m;        // Nótese que los números viajan en objetos de clase MensajeEntero.

   cout << "PROBLEMA : RECONOCIMIENTO DE NUMEROS PRIMOS\n\n";
   cout << "Cantidad de datos a reconocer ? (menos de 50) : ";
   cin  >> cantDatos;
   if (cantDatos > 0)
      for (i=0; i < cantDatos; i++)
      {   m.escribeContenido(2+i);
          adiciona(m);
      }
}
```

```
void DatosEntPrimo::exhibe (void)
{  int i;

  if (cantDatos <= 0)
    cout << "Ningun dato para reconocer !!\n";
  else
  {  cout << "Datos de entrada : ";
     vista();
     cout << "\n";
  }
}
```

Exhibición de los números primos encontrados :

```
class DatosSalPrimo : public DatosSalPipe
{  public:
       DatosSalPrimo(int capacidad) : DatosSalPipe(capacidad) {}
         void exhibe (void);
};


void DatosSalPrimo::exhibe (void)
{  int i;

   cout << "SOLUCION : \n";
   cout << "Numeros primos encontrados : ";
   if (existeElem())
   {  vista();
      cout << "\n";
   }
   else
      cout << "No se encontro ningun numero primo";
}
```

Solución del problema de reconocedor de números primos con el paradigma de *pipes*

```
main()
{
   DatosEntPrimo infoEnt(50);
   DatosSalPrimo infoSal(50);
   clrscr();

   infoEnt.introduce();          // Captura los datos de entrada y
   infoEnt.exhibe();             // los muestra.

   FiltroPrimo filPrimo;         // Declara el objeto que especializa al paradigma de flujo entre Pipes.
   ProblemaPipe primo(infoEnt, infoSal, filPrimo);   // Declara el Problema con respecto al reconocedor
                                                     // de números primos.
   Null    dos;
   infoSal.exhibe();             // Muestra los resultados.
}
```

3. Mecanismo de interacción *buffer*

El *buffer* es un mecanismo que temporalmente guarda en un depósito de cierta capacidad, los mensajes enviados por un emisor, liberando a éste de esperar hasta que un receptor se encuentre listo para acudir a tomar cada mensaje. Con la ayuda de este mecanismo se puede convertir la comunicación síncrona en asíncrona; no tiene sentido en esta última, dado que los canales asíncronos implícitamente ofrecen la capacidad de almacenamiento de mensajes.

3.1. Modelo OO del mecanismo *buffer*

La clase *Buffer* es el modelo OO del mecanismo de interacción *buffer* descrito en el párrafo anterior (ver el diagrama de clases de la Fig. 6.16); la siguiente figura es su representación gráfica y los puntos subsecuentes presentan los principales rasgos de la clase :

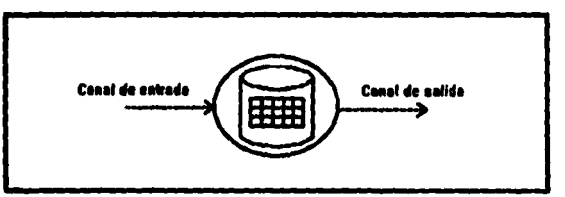

Fig. 6.18 Símbolo del mecanismo Buffer.

a. Buffer es clase heredera de EmisorReceptor, por tanto origina objetos activos.

b. Buffer tiene asociados dos canales, uno de entrada, por donde recibe los mensajes del emisor, y otro de salida, por donde se los entrega al receptor.

d. El depósito del *Buffer* puede ser implementado de diferentes maneras, una de ellas es la cola cuya política FIFO es la más socorrida en los esquemas de despacho. El buffer modelado define el depósito de la misma manera que en el canal asíncrono[6].

[6] Ver capítulo anterior : modelo OO de *canal asincrono*.

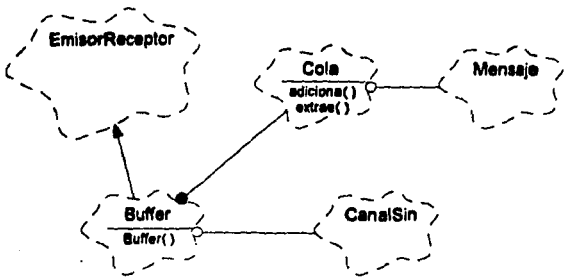

Fig. 6.16 Diagrama de clases de del mecanismo 'Buffer'.

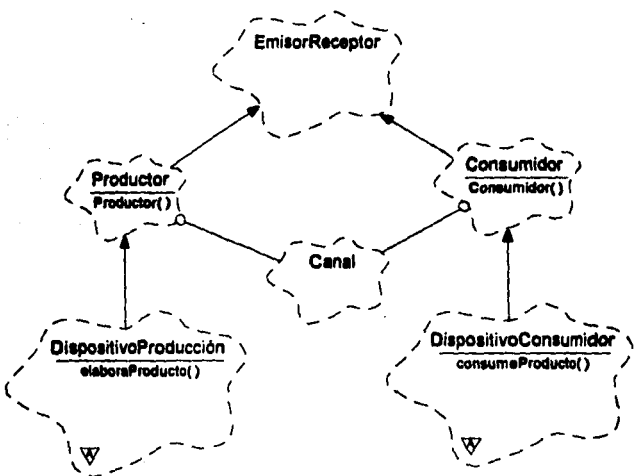

Fig. 6.17 Diagrama de clases del paradigma de interacción 'Productor-Consumidor'.

3.2. Seudocódigo de la clase *Buffer*

```
class Buffer : public EmisorReceptor
{
  private:
      Cola[Mensaje*] deposito;          // Se construye vacio
      Mensaje        *m;
  public:
      Buffer (CanalSin*, CanalSin*);    // Son canales síncronos.
}
```

El comportamiento de ***Buffer*** (definido en su función constructora), se centra en revisar si hay un emisor o un receptor solicitando ser atendido; de haber un emisor y existir espacio libre en su depósito, recibe el mensaje y lo almacena; de haber un receptor constata que existan mensajes almacenados, siendo así toma uno y lo envía al receptor. Si las condiciones son favorables tanto para atender al emisor como al receptor (ambos pendientes en sus respectivos canales), elige al azar a quien atender.

```
Buffer::Buffer (CanalSin* cEnt, CanalSin* cEnt)
{
  do
  { select
    {     when (cEnt->emisoresPendientes() && deposito.existeEspacio())
          {     recibe(m,cEnt);
                deposito.adiciona(m);
          }
          when (cSal->receptoresPendientes() && deposito.existeElem())
          {     m = deposito.extrae();
                envia(m,cSal);
          }
    }
  } while (m->leeId() != MSJ_CTRLFIN
}
```

4. Paradigma de interacción *productor - consumidor*

Productor y *consumidor* son procesos complementarios. El productor genera por algún medio elementos que requiere el consumidor. El consumidor utiliza esos elementos de acuerdo a sus propias necesidades. Los elementos deben viajar del productor al consumidor, a través de algún medio de comunicación, si responde a un modelo síncrono o asíncrono es indiferente, el paradigma es útil en ambos casos.

Generalmente la velocidad de producción y la de consumo no son iguales, por tanto, es conveniente que la interacción se de vía un intermediario de almacenamiento :

- Si la comunicación es asíncrona, el problema está resuelto por la propia naturaleza del canal asíncrono. Un canal asíncrono conecta directamente al productor y al consumidor. Graficamente se ve así :

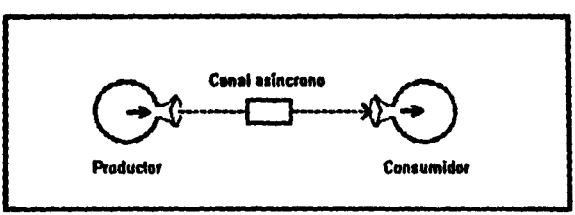

Fig. 6.19 Interacción *Productor-Consumidor* vía un canal asíncrono.

- Si la comunicación es síncrona, es preciso emplear un buffer como proceso intermediario, conectado por un lado con el productor y por otro, con el consumidor, es decir :

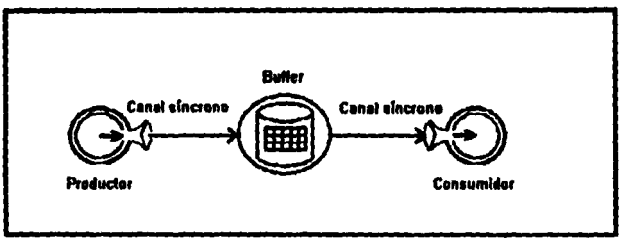

Fig. 6.20 *Productor-Consumidor* vía canales síncronos y un *Buffer*.

4.1. Modelo OO del paradigma de interacción productor- consumidor

Los procesos productor y consumidor se traducen en las clases Productor y Consumidor, respectivamente (ver el diagrama de clases en la Fig. 6.17). Los siguientes puntos las describen :

a. Ambas clases forman objetos de actividad concurrente y son herederas de la clase *EmisorReceptor*.

b. Productor tiene un canal de salida por donde envía los productos que genera, Consumidor presenta un canal de entrada, por donde los recibe.Cada producto corresponde a un tipo de mensaje particular, que para guardar consistencia en la comunicación, es una clase heredera de la clase Mensaje.

c. Productor y Consumidor particularizan su esquema general de comportamiento mediante funciones virtuales[7].

[7] Aplícase nuevamente el problema citado en la forma de especialización de mecanismos.

4.2. Seudocódigo de las clases del paradigma Productor - Consumidor

```
class DispositivoProduccion
{ public:
        iniciaProduccion()    {}
        elaboraProducto() = 0;
        terminaProduccion() {}
};

class Productor : EmisorReceptor
{ private:
        Mensaje   *pdto;
  public:
        Productor (Canal*, DispositivoProduccion&);
};


Productor::Productor (Canal* cProduccion, DispositivoProduccion& dispoProd)
{
  dispoProd.iniciaProduccion ();
  do
  { dispoProd.elaboraProducto (pdto);
    envia (pdto,cProduccion);
  } while (pdto->leeId () != MSJ_CONTROL);
  dispoProd.terminaProduccion ();
}


class DispositivoConsumo
{ public:
        iniciaConsumo()    {}
        consumeProducto() = 0;
        terminaConsumo() {}
};


class Consumidor : public EmisorReceptor
{ private:
        Mensaje *pdto;
  public:
        Consumidor (Canal*, DispositivoConsumo&)
};


Consumidor::Consumidor (Canal* cConsumo, DispositivoConsumo& dispoCons)
{
  dispoCons.iniciaConsumo ();
  do
  { recibe (pdto,cConsumo);
    dispoCons.consumeProducto (pdto);
  } while (pdto->leeId () != MSJ_CONTROL)
  dispoCons.terminaConsumo ();
}
```

4.3. Ejemplo de aplicación : impresión del código ASCII de los datos de un archivo.

En este ejemplo, el productor lee caracteres de un archivo y los envía al consumidor, éste obtiene su representación en código ASCII y los imprime. El consumidor puede verse como el modelo de una impresora, con funciones para dar formato a la información recibida.

4.3.1. Modelo OO del problema de aplicación

Dado que la mayoría de las clases están predefinidas, es únicamente necesario definir las clases de *dispositivos de especialización* de *Productor* y *Consumidor* (ver el diagrama de clases del ejemplo en la Fig. 6.21 y el diagrama de objetos, en la Fig. 6.22).

4.3.2. Seudocódigo de la solución

```
clase DispoProdCaracter : DispositivoProduccion
{ private:
        FILE        *arch;
        char        nomArch[30];
        MsjCaracter *caracter;
  public:
        void iniciaProduccion ();
        void elaboraProducto (Mensaje* pdto);
        void terminaProduccion ();
}


void DispoProdCaracter::iniciaProduccion(void)
{
     // Solicita el nombre del archivo con el que se va a trabajar.
     // Abre el archivo y prepara arch como un apuntador a él.
}


void DispoProdCaracter::elaboraProducto (Mensaje* pdto)
{ char car;

   car = arch->lee();                    // Lee un caracter del archivo apuntado por arch.
   if (car == EOF)
      pdto = new MensajeControl();  // Heredero de Mensaje con identificador MSJ_CTRL.
   else
   { caracter = new MsjCaracter();    // Mensaje con su contenido de tipo caracter.
     caracter->escribeContenido(car);
     pdto = caracter;
   }
}
```

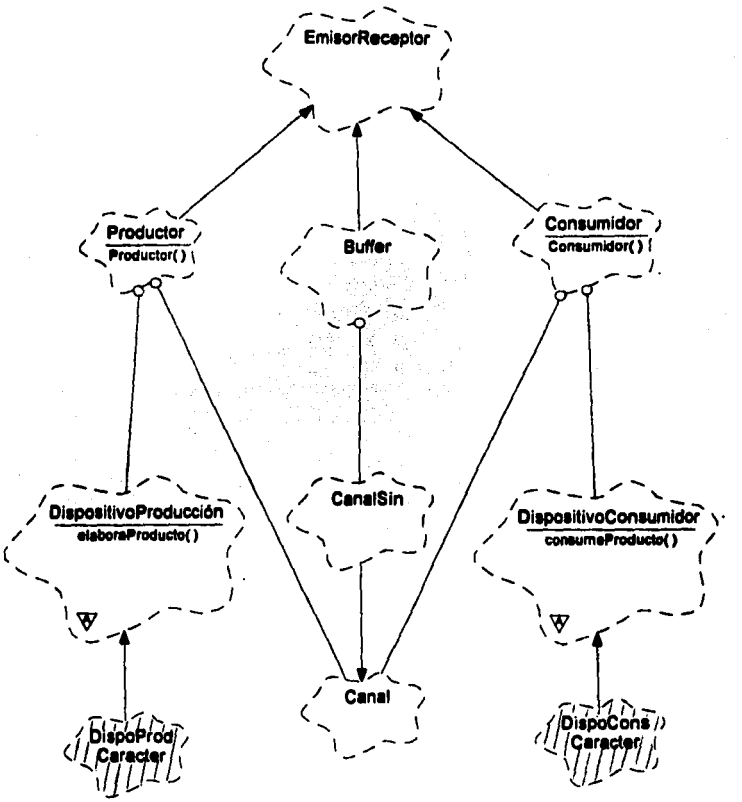

Fig. 6.21 Diagrama de clases del ejemplo de aplicación del mecanismo 'Buffer' y del paradigma de interacción 'Productor-Consumidor' : "Impresión del código ASCII de los datos de un archivo".

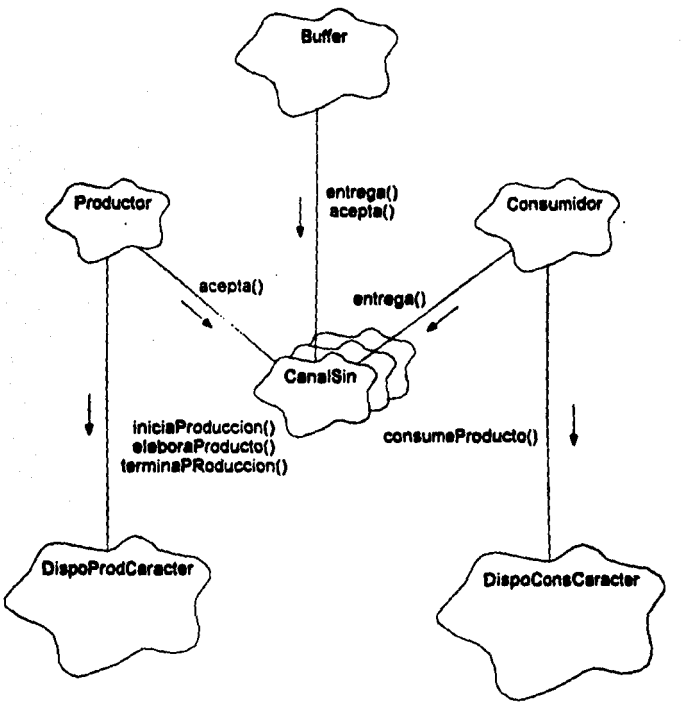

Fig. 6.22 Diagrama de objetos del ejemplo de aplicación del mecanismo 'Buffer' y del paradigma de interacción 'Productor-Consumidor' : "Impresión del código ASCII de los datos de un archivo".

```
void DispoProdCaracter::terminaProduccion(void)
{
      // Cierra el archivo abierto.
}


class DispoConsCaracter :  DispositivoConsumo
{
  private:
        MajCaracter  *caracter;
  public:
        void consumeProducto (Mensaje* pdto);
}


void DispoConsCaracter::consumeProducto (Mensaje* pdto)
{
      // Lee el caracter que viaja en pdto.
      // Obtiene el código ASCII de ese caracter y lo imprime.
}
```

Solución con *canal asíncrono* :

```
main ()
{
  CanalAsincrono    intermediario;

  Productor (&intermediario);
  Consumidor (&intermediario);
}
```

Solución con *canal síncrono* :

```
main()
{
  CanalSincrono        cProduccion, cConsumo;

  DispoProdCaracter   dispoProd;
  DispoConsCaracter   dispoCons;

  Productor (&cProduccion,dispoProd);
  Buffer (&cProduccion, &cConsumo);
  Consumidor (&cConsumo,dispoCons);
}
```

5. Mecanismo de interacción *difusor*

El mecanismo ***difusor*** es un propagador de mensajes. Manda una copia del mensaje que recibe por su canal de entrada a los canales de salida. La difusión se realiza a todos o sólo a algunos de los canales de salida, dependiendo de la condición de difusión que posea el mecanismo.

5.1. Modelo OO del mecanismo *difusor*

La clase ***Difusor*** (modelo OO de *difusor*) puede apreciarse simbólicamente en la Fig. 6.23 y jerárquicamente en la Fig. 6.24 :

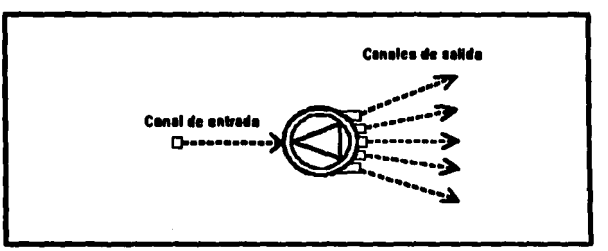

Fig. 6.23 Símbolo del mecanismo *Difusor*.

Aparentemente el *Difusor* puede ser confundido con un canal uno a muchos; sin embargo existe una diferencia radical entre ellos : el primero, por cada mensaje que recibe envía tantas copias como procesos deban ser notificados por ese mensaje, el segundo, transmite el mensaje nada más al primer proceso que acuda a recibirlo. Adicionalmente :

a. El mecanismo *Difusor* es un proceso y como tal, heredero de la clase ***EmisorReceptorConex***. Podría ser modelado como un mecanismo sin control de conexiones; pero de ser así, la condición de difusión no puede considerar la identidad ni número real de los procesos que recibirán la difusión del mensaje.

b. *Difusor* tiene un canal de entrada y una serie de canales de salida. Estos últimos la mayoría de las veces de tipo *CanalConex_1x1*, para tener la seguridad de que cada mensaje divulgado será recibido por cada uno de los procesos conectados como destinos y no más de una vez.

c. Lo ideal es que cada mensaje llegado al *Difusor* sea simultáneamente entregado a todos los canales de salida, para ello se necesitaría una instrucción en paralelo[8]. El modelo actual, responde a los recursos del lenguaje de implantación : la entrega es secuencial y para disminuir la posibilidad de espera tanto del *Difusor* como de los procesos destino, se recomienda el uso de canales asíncronos.

d. No es preciso que *Difusor* saque explícitamente copias para cada destino, al ser heredero de *EmisorReceptorConex*, puede utilizar la operación *envia(...)*, misma que normalmente saca la copia del mensaje recibido como parámetro.

e. La condición de difusión de un mensaje puede ser pactado de modo particular como se explicó en los mecanismos anteriores. La clase que encierra la interfaz para la condición de difusión es *CondicionDifusion*. Si no se define ninguna especialización, el *Difusor* debe divulgar sin restricciones.

6.2. Seudocódigo de la clase *Difusor*

```
clase Difusor : EmisorReceptorConex
{       Mensaje *m;
  public:
        Difusor (CanalConex*, Cola[CanalConex*], CondicionDifusor&)
};


Difusor::Difusor (CanalConex* cnlEnt, Cola[CanalConex*] cnlsSal, CondicionDifusor& dif)
{  int i, lim;

   conectaCanalEntrada (cnlEnt);
   for (i = 0, lim = cnlsSal.medida();  i < lim; i++)
        conectaCanalSalida (cnlsSal[i])    // [i] devuelve el elemento i-ésimo de la cola.
   do
   {  recibe (m,cnlEnt);
        for (i = 0, lim = canalesSalida.medida(); i < lim; i++)
            if (dif.cumpleCondDifusion(m,canalesSalida[i])
                envia(m,canalesSalida[i]);
   } while (m->leeId() <> MSJ_CONTROL);
   desconectaCanalEntrada (cnlEnt);
   for (i = 0; i < lim; i++)
        desconectaCanalSalida (canalesSalida[i]);
}
```

[8] Al estilo de PAR, en OCCAM.

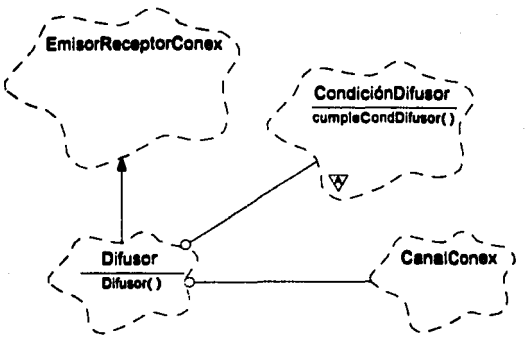

Fig. 6.24 Diagrama de clases del mecanismo 'Difusor'.

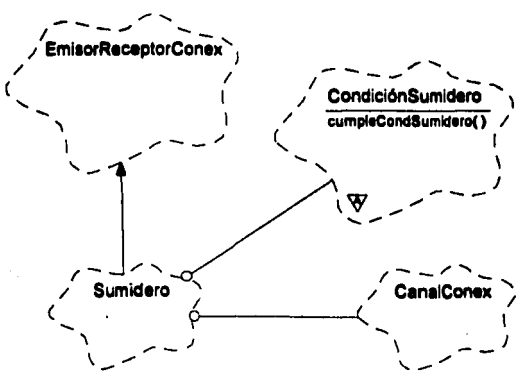

Fig. 6.25 Diagrama de clases del mecanismo 'Sumidero'.

La condición de difusión puede establecerse en base al mensaje recibido y/o al canal de entrega, además, siendo que el canal es definido con control de conexiones, también en relación con la identidad y número de procesos destino.

```
class CondicionDifusor
{
  virtual BOOL cumpleCondDifusion (Mensaje* m, CanalConex* c) { return TRUE; }
        // Al devolver TRUE, se indica que 'm' debe ser entregado a 'c'.
        // En este nivel de, se autoriza la difusión de 'm' incondicionalmente.
};
```

6.3. Ejemplo de aplicación del mecanismo *Difusor*: "Difusión-absorción de mensajes a varios procesos"

Se amplia el ejemplo de aplicación de *puente*. Allí el *procesoA* envía mensajes 'por siempre', ahora, piénsese que sólo quiere enviar 100 mensajes y luego terminar, de igual manera, el resto de procesos deben concluir cuando dejan de ser útiles. Así mismo, en el ejemplo mencionado, los mensajes enviados por el *procesoA* son tomados por el *puente1* o el *puente2* (o exclusivo), ahora se pretende que ambos puentes reciban una copia de cada mensaje, para lo cual se necesita un mecanismo (un *difusor*) que multiplique los mensajes del *procesoA*, haciendo llegar una copia a cada *puente*. La necesidad de multiplicar los mensajes se hace patente principalmente si se piensa en el mensaje de finalización, mismo que debe llegar a todos los procesos.

Modelo OO de la solución (1ra. parte)

Si el *ProcesoA* envía un mensaje por el *'canal_1'*, lo toma el *Puente 1* ó el *Puente 2*, uno u otro porque están conectados a través de un CanalAsinConex_1xn. Si el *ProcesoA* manda dos mensajes de finalización, no resuelve el problema porque puede darse el caso que sea uno de los *Puentes* (el más diligente), quien reciba ambos. El problema se soluciona agregando un *Difusor* a la red de procesos. Como fuente del *Difusor* se conecta al *ProcesoA* y como destinos, a los *Puentes 1* y *2*. De modo gráfico se puede observar en el siguiente cuadro (en él también se incluye el mecanismo *Sumidero* que se explicará posteriormente) :

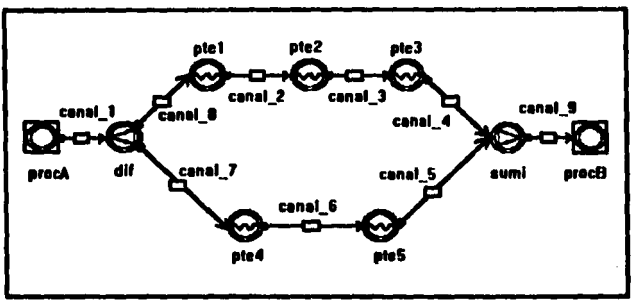

Fig. 6.26 Difusión-absorción de mensajes mediante un Difusor y un Sumidero.

Ahora bien, surge otro problema, al ProcesoB le llegan dos mensajes por cada mensaje enviado por el ProcesoA. No existe problema en cuanto a los mensajes de datos se refiere, todos son útiles ya que cada uno tiene un contenido diferente formado por las máquinas que atraviesa, el conflicto surge con los mensajes repetidos, un ejemplo de ésto lo presentan los mensajes de finalización : al ProcesoB le envían dos mensajes de finalización, uno del *Puente 3* y otro del *Puente 4*; al recibir el primer mensaje concluirá, ignorando el segundo, estos mensajes repetidos se convierten en basura, cuya cantidad, dependiendo de la aplicación puede ser extensa y de consecuencias indeseables, pudiendo dejar en espera infinita a los procesos dueños de mensajes de finalización diferentes al primero (cuanto más si los canales son síncronos). Para remediar este conflicto se introduce un nuevo mecanismo: el *Sumidero*.

6. Mecanismo de interacción *sumidero*

Recibe mensajes de varios orígenes y sólo deja pasar aquellos que cumplen la condición del sumidero.Puede ser visto como la contraparte del mecanismo difusor, mientras éste propaga o multiplica un mensaje, éste contrae o absorbe los mensajes recibidos.

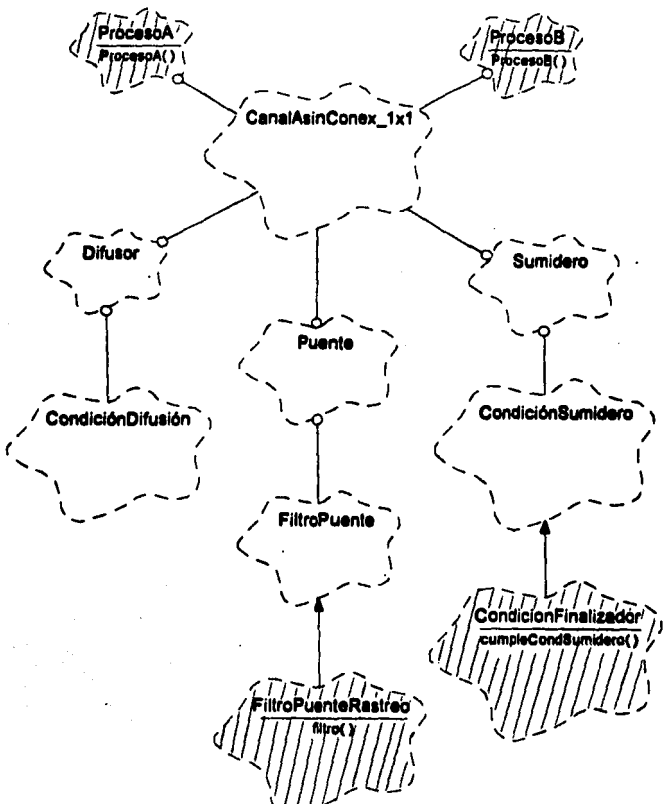

Fig. 6.27 Jerarquía de clases del ejemplo de aplicación de losmecanismos 'Difusor' y 'Sumidero' : "Difusión-absorción de mensajes".

6.1. Modelo OO del mecanismo *sumidero*

El proceso *sumidero* visto como clase (llamada ***Sumidero***) puede observarse en el diagrama jerárquico de la Fig. 6.25; paralelamente, su representación simbólica, en la siguiente figura :

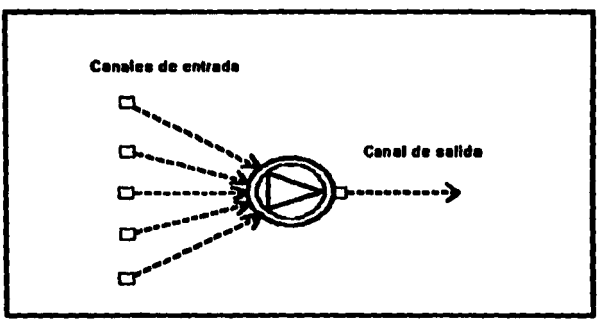

Fig. 6.28 Símbolo del mecanismo *Sumidero*.

Sumidero tiene las características dadas en los próximos incisos :

a. ***Sumidero*** es clase heredera de ***EmisorReceptorConex***.

b. Tiene asociados una serie de canales de entrada con conexiones (como en Difusor, para sus canales de salida, se recomienda aquí para los de entrada, el empleo de canales asíncronos para evitar en lo posible las esperas), y cualquier variedad de *CanalConex* como canal de salida .

c. Especifica una condición de absorción de mensajes, a través de una función virtual que se especializa para cada problema de aplicación. Esta función, como en mecanismos explicados anteriormente es parte de una clase auxiliar.

d. ***Sumidero*** recorre cada canal de entrada para recibir los mensajes a analizar. Dado que algunos canales pueden ser de tipo muchos a uno, se reunen en él varios mensajes y por justicia, se atiende a cada canal con tantas recepciones como fuentes conectadas tenga. En estos casos para conocer la identidad exacta del proceso fuente, el mensaje debería acarrear la información. Cuando los canales son de tipo uno a uno, este conocimiento es directo a través del propio canal.

e. Si *Sumidero* contiene un sólo canal de entrada puede verse parecido a un proceso ***Puente***, aunque no idéntico.

6.2. Seudocódigo de la clase *Sumidero* y sus clases asociadas

```
class Sumidero : EmisorReceptorConex
{
  public:
      Sumidero (Cola[CanalConex*], CanalConex*, CondicionSumidero&)
}

Sumidero::Sumidero (Cola[CanalConex*] calsEnt, CanalConex* calSal,
                    CondicionSumidero& sum)
{ BOOL continua = TRUE;
  unsigned     i;

  for (i = 0, lim = calsEnt.medida(); i < lim; i++)
      conectaCanalEntrada (calsEnt[i]);
  conectaCanalSalida (calSal);
  do
  { for (i = 0, lim_i = canalesEntrada.medida(); i < lim; i++)
        recibe (m,canalesEntrada[i]);
        if (sum.cumpleCondSumidero (m, canalesEntrada,i, continua)
            envia (m,calSal);
  } while (continua);
  for (i = 0; i < calsEnt.medida(); i++)
      desconectaCanalEntrada (calsEnt[i]);
  desconectaCanalSalida (calSal);
}
```

La condición de absorción por parte del Sumidero puede considerar el mensaje recibido y/o el canal por el que lo recibe. Del canal, de ser necesario, es posible sustraer el conocimiento del proceso fuente del que proviene el mensaje. Para determinar algunas condiciones de absorción, es también conveniente conocer el número total de fuentes conectadas[10].

```
class CondicionSumidero
{
  virtual BOOL cumpleCondSumidero (Mensaje*, Cola[CanalConex*], unsigned, BOOL&)
              { return TRUE; }
          // Al devolver TRUE, se indica que se dejara pasar a 'm'.
          // En este nivel de, se autoriza el paso por el Sumidero incondicionalmente.
}
```

[10] Un ejemplo de ésto se verá más adelante.

6.3. Ejemplo de aplicación del mecanismo *Sumidero* : "Difusión - absorción de mensajes a varios procesos"

El ejemplo de aplicación de puente, retomado después para difusor, ahora se completa aplicando un sumidero. El conflicto está en el ProcesoB al que le envían más de un mensaje de finalización. La solución consiste en dejar pasar nada más un mensaje de este tipo, cuando se esté seguro que no se recibirán más.

6.3.1. Modelo OO de la solución (2da. parte)

La solución implica únicamente la definición de la condición del sumidero, ésto en la clase *CondicionFinalizador*, heredera de *CondiciónSumidero* (ver las figuras 6.27 y 6.29).

Al declarar la red de procesos para este problema, ahora se incluye un objeto *Sumidero* instanciado con la clase *CondicionFinalizador*. Los canales *'canal_4'* y *'canal_5'* se conectan como sus canales de entrada y un nuevo *CanalAsinConex_1x1*, llamado *'canal_9'*, como su salida, mismo que ahora será la entrada del *ProcesoB*. Gráficamente se puede apreciar en la Fig. 6.26.

6.3.2. Seudocódigo de la solución (total)

La codificación de la solución se reduce a :

```
class CondicionFinalizador : CondicionSumidero
{
  private:
      unsigned  noFinales;
  public:
      CondicionFinalizador () { noFinales = 0; }
      BOOL cumpleCondSumidero (Mensaje*, Cola[CanalConex*], unsigned, BOOL&)
};


BOOL CondicionFinalizador::cumpleCondSumidero (Mensaje* m, Cola[CanalConex*] calsEnt,
                                               unsigned calActual, BOOL& continua)
{   if (m->leeId() <> MSJ_CONTROL)
        return TRUE;          // Deja pasar el mensaje si no se trata de un marcador de finalización.
    else
        if (++noFinales >= calsEnt.medida())  // Si el # de los mensajes de fin recibidos es igual al # de
            continua = FALSE;                 // fuentes, quiere decir que todos concluyeron y por tanto el
            return TRUE;                      // Sumidero deja pasar el mensaje de fin y termina.
        else                                  // Absorbe los mensajes de fin hasta que todas las fuentes
            return FALSE;                     // hayan enviado el suyo.
}
```

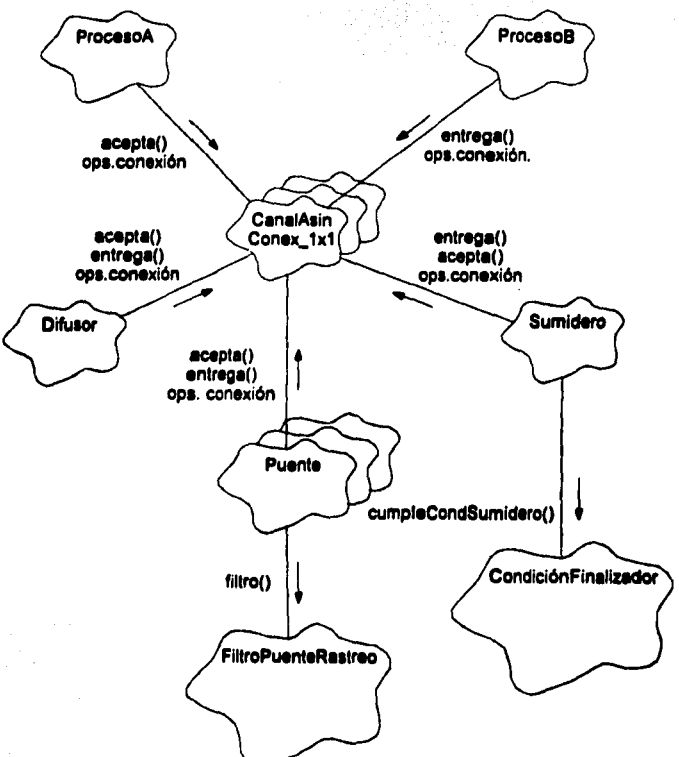

Fig. 6.29 Diagrama de objetos del ejemplo de aplicación de los mecanismos 'Difusor' y 'Sumidero' : "Difusión-absorción de mensajes".

```
main ()  // Declaración de la solución.
{
  CanalAsinConex_1x1      canal_1, canal_2, canal_3, canal_4, canal_5,
                          canal_6, canal_7, canal_8, canal_9;

  ProcesoA                procA (&canal_1);

  CondicionDifusor        condDif;      // Condición default de difusión.
  Cola[CanalConex*]       colaSalDif;
                                  // Introducir 'canal_7' y 'canal_8' en la cola 'colaSalDif'
  Difusor                 dif (&canal_1,colaSalDif, condDif);

  FiltroPuenteRastreo     fil;          // Condición default de filtraje de Puente.
  Puente                  pte1(&canal_8,&canal_2, fil), pte2(&canal_2,&canal_3, fil),
                          pte3(&canal_3,&canal_4, fil), pte4(&canal_7,&canal_6, fil),
                          pte5(&canal_6,&canal_5, fil);

  Cola[CanalConex*]       colaEntSumi;
                                  // Introducir 'canal_4' y 'canal_5' en la cola 'colaEntSumi.
  CondicionFinalizador    condSumi;   // Condición para absorver los mensajes de Fin.
  Sumidero                sumi(colaEntSumi,&canal_9,condSumi);

  ProcesoB                procB (&canal_9);
}
```

7. Descripción del proceso propagador

La propagación general de mensajes en una red de procesos- entendiéndose como tal el hacer del conocimiento de todos los nodos el contenido de los mensajes-, es una función necesaria y frecuente. Esto se puede lograr insertando a lo largo de toda la red varios *difusores* y *sumideros* (dependiendo de la topología de la red de procesos); sin embargo, la tarea puede ser complicada y lo que es peor provocar sobrecarga innecesaria en el sistema, pues debe recordarse que los mecanismos *difusor* y *sumidero* son entes activos, lo cual no significa que dichos mecanismos sean inservibles dado que son adecuados para la difusión y absorción de mensajes a nivel local, aunque sí quiere decir que no son recomendables para la propagación a través de toda la red, para cuyo caso se propone una nuevo mecanismo : el *propagador* (pensado de modo teórico, simplemente como una sugerencia a la crítica del uso extendido de los *difusores* y *sumideros*, y como aplicación de los conceptos de uniformidad y extensibilidad del modelo de mecanismos de interacción propuestos en este trabajo de tesis).

7.1. Modelo OO del proceso *propagador*

El propagador es una variedad de proceso *emisor/receptor* (específicamente de la clase *EmisorReceptorConex*, ver la Fig. 6.32), que puede iniciar la propagación de un mensaje o puede recibir el mensaje de algún otro proceso y hacerlo llegar a los procesos que están conectados a él (ver la Fig. 6.31 para reconocer el símbolo de *Propagador*).

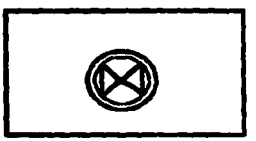

Fig. 6.30 Símbolo de Propagador.

El problema central en la propagación de mensajes está en el tratamiento de los mensajes repetidos. Para evitar que un proceso reciba el mismo mensaje, a partir de varias fuentes, se puede seguir la directriz de un árbol abarcador (con nodos igual a los procesos y aristas igual a los canales que los unen).

El proceso que inicia la propagación de un mensaje debe calcular el árbol abarcador que cubra la red (para que cada usuario aplique el método que le parezca más eficiente, este cálculo corresponde a una función diferida); los demás procesos reciben el mensaje en propagación, conocen su contenido y de acuerdo al árbol abarcador que también recibieron, lo pasan a los procesos que tienen conectados y que están comprendidos en el árbol. Por tanto, cada mensaje a ser propagado es de una clase especial que además de los datos de un contenido normal (entero, caracter, gráfico, etc.), incluye el arból abarcador.

7.2. Seudocódigo de la clase *Propagador*

```
class Propagador : EmisorReceptorConex
{       MajPropagacion    mPropaga, *mPaso;
        ArbolAbarcador    *arbolAbarca;
  protected:
        void iniciaPropagacion(ArbolAbarcador*  arbol = NULL);
        void propaga(MajPropagacion* mb);
        virtual ArbolAbarcador* calculaArbolAbarcador()
             { /*  Posiblemente algún método por omisión  */ }
};


void iniciaPropagacion(ArbolAbarcador*  arbol = NULL)
{    if (arbol ==NULL)
        arbolAbarca  = calculaArbolAbarcador();
     else
        arbolAbarca =. arbol;
     mPropaga.escribeArbol(arbolAbarca );
}


void propaga(MajPropagacion *mProp)
{    if (arbolAbarca->compara(mProp->leeArbolAbarcador()))    // TRUE si debe actualizar
        arbolAbarca = mProp->leeArbolAbarcador();
     for (i= 0; i < canalesSalida.medida(); i++)
        if (arbolAbarca.incluye(canalesSalida[i])
             envia(mProp,canalesSalida[i]);
}
```

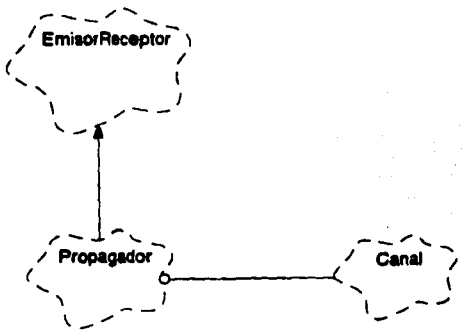

Fig. 6.31 Diagrama de clases del proceso 'Propagador'.

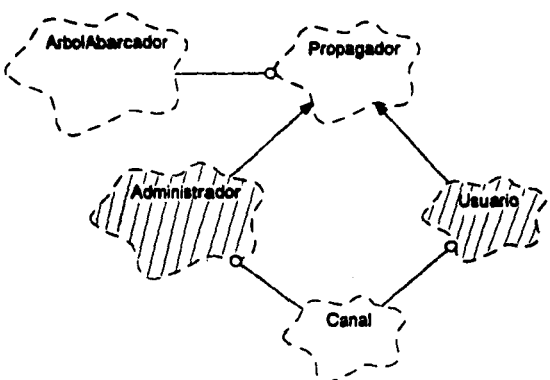

Fig. 6.32 Diagrama de clases del ejemplo de aplicación del proceso 'Propagador' : "Propagación de mensajes por la red".

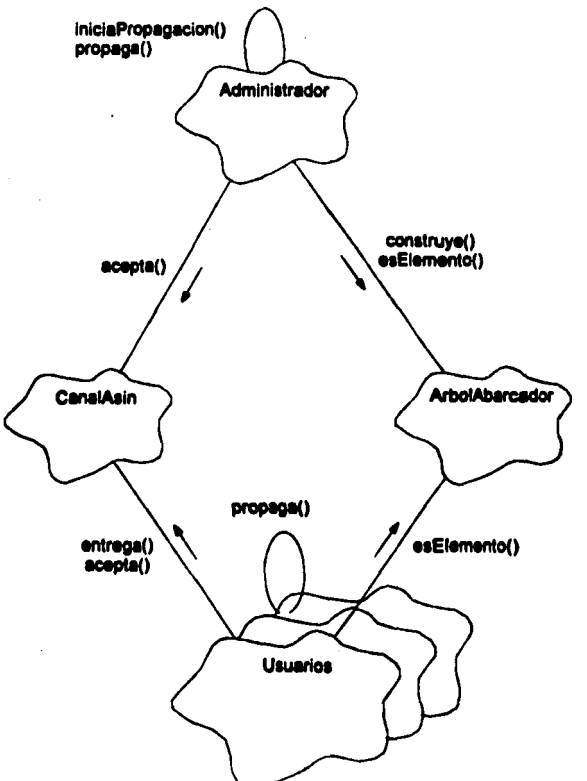

Fig. 6.33 Diagrama de objetos del ejemplo de aplicación del proceso 'Propagador' : "Propagación de mensajes por la red".

7.3. Ejemplo de aplicación : "Propagación de mensajes por todos los proceso de la red"

El proceso *administrador* de la red desea informar continuamente a los procesos *usuarios* de la red sobre diferentes anuncios, por ejemplo sobre nuevos servicios disponibles, posibles fallas de algunos dispositivos, próxima suspensión del sistema, etc.

7.3.1. Modelo OO de la solución

De todos los procesos en la red, uno de ellos es descendiente de la clase *Administrador* y tendrá las funciones de un agente distinguido en la red, el resto de procesos son instancias de la clase *Usuario*, con diferentes clases de especialización (ver la Fig. 6.32 y 6.33).

7.3.2. Seudocódigo de la solución

```
class Administrador : public Propagador
{
  char    nota[TAM_ANUNCIO];

  public:

    Administrador (Cola[CanalConex*] colaEnt, Cola[CanalConex*] colaSal)
    {    // Conecta los canales que le comunican con sus procesos vecinos.
         while (TRUE)
         {    // Prepara el anuncio que quiere enviar a todos los procesos usuarios.
              // Introduce en 'nota' la información que desea propagar.
              mPropaga.escribeContenido(nota);
              iniciaPropagacion();
              propaga(mPropaga);
              // Otras actividades del Administrador.
         }
         // Desconecta los canales que le comunican con sus procesos vecinos.
    }
};
```

```
class Usuario : public Propagador
{
    Usuario (Cola[CanalConex*] cnlsEnt, Cola[CanalConex*] cnalSal,
             EspecialidadUsuario& esp)
          // Conecta los canales que comparte con sus vecinos.
    while (TRUE)
      for (i = 0; i < canalesEntrada.medida(); i++) [11]
      {    recibe(mPaso,canalesEntrada[i]);
           if (mPaso->leeId() == MSJ_PROPAGA)
              propaga(mPaso);
           // Consume la información recibida, de acuerdo a la clase EspecialidadUsuario,
           // que especializa al proceso de la misma manera que los dispositivos creados
           // con este propósito en los mecanismos explicados previamente.
      }
    // Desconecta sus canales.
};


main ()
{
    // El lanzamiento de los procesos en un programa está compuesto por una red
    // de procesos formada por un Administrador y 'n' Usuarios, compartiendo canales
    // con conexiones en cualquier topología.
}
```

[11] Una vez más se lamenta la ausencia del comando al estilo ALT de OCCAM.

CONCLUSIONES

El objetivo que dirigió este trabajo está intrínsecamente ligado a la **reusabilidad**, a la reducción del trabajo de los implementadores de programas distribuidos, a través del empleo de mecanismos estandar de interacción predefinidos. El modelo en el que se desarrollan estos mecanismos impone características propias de reusabilidad, con menor o mayor flexibilidad a su adecuación en problemas particulares. El modelo Orientado a Objetos nos permitió crear esas componentes en una organización coherente u homogénea, es decir en jerarquías que factorizan en los niveles básicos las características comunes a todos los mecanismos y en niveles superiores, las caracteríticas que los enriquecen o especializan; pero como todos ellos responden a un modelo único, se obtienen principalmente las siguientes ventajas:

- Configurar y reconfigurar la topología de los procesos, colocando o sustituyendo los canales que los conectan y/o insertando los procesos correspondientes a mecanismos de interacción de mayor nivel que los canales, como si se armara un artefacto conformado por piezas fácilmente removibles, gracias a su relación de parentesco en la jerarquía.

- Homogeneidad en los nombres, invocación y funcionamiento de las funciones de las componentes, no importando sus labores específicas, como consecuencia del uso amplio del polimorfismo y el ligado dinámico.

El siguiente elemento a considerar en la concepción de las componentes, después del modelo en el que se desarrollan, son las herramientas de implementación; en el caso de problemas concurrentes/distribuidos enfocados con la metodología OO, cuanto más resulta conflictivo el elegir un lenguaje de programación, ya que son determinantes sus características para facilitar u obstaculizar la explotación de reusabilidad y extensibilidad. En este trabajo se experimentó con una versión concurrente de C++. A raíz de las pruebas realizadas, se corroboraron anomalías con herencia múltiple, herencia repetida y redefiniciones innecesarias (sobre todo para calificación de funciones ambiguas). Algunas anomalías provienen del lenguaje C++; otras de la extensión concurrente -como el explicado en el capítulo 6-; y otras de la combinación de ambos.

PROYECCIONES FUTURAS :

- Incorporar nuevos mecanismos, por ejemplo canales bidireccionales.
- Implantar estos mecanismos en un ambiente físicamente distribuido.
- Crear un ambiente gráfico en el que se tenga un menu de los símbolos de los mecanismos de interacción, desde los más simples a los más complejos de la jerarquía. Este ambiente permitirá construir interactivamente (por ejemplo con un mouse) la topología de procesos de las aplicaciones. La aplicación creada bajo este ambiente puede traducirse en programas en funcionamiento real en la red, o bien en una simulación para fines de prueba o de enseñanza, ya que puede derivar en una adecuada herramienta didáctica que visualice el trabajo de diferentes modelos de sincronizacion y comunicación.

REFERENCIAS

[Albarrán, *et.al.* 89]
M. Albarrán , S. López , H. Oktaba, V.G. Sánchez. *Sistemas Distribuidos en redes y sistemas operativos heterogéneos.* Propuesta del Proyecto de Investigación para la DGAPA. Reporte Interno. 1989.

[Andrews 91:a]
Gregory R. Andrews. *Paradigms for Process Interaction in Distributed Programs,* ACM Computing Surveys, Vol. 23, No. 1, March 1991.

[Andrews 91:b]
Gregory R. Andrews. *Concurrent Programming. Principles and Practice,* The Benjamin/Cummings Publishing Company Inc., 1991.

[Auerbach, *et.al.* 92]
J. S. Auerbach, *et. al.* (12 autores : IBM T. J. Watson Research Center NY). *High - Level Language Support for Programming Distributed Systems,* IEEE 1992.

[Bal, *et.al.* 93]
Henri E. Bal, Jennifer G. Steiner, Andrew S. Tanenbaum. *Programming Languages for Distributed Computing Systems.* ACM Computing Surveys, Vol. 21, No. 3, September 1989.

[Ben-Ari 90]
M. Ben-Ari. *Principles of Concurrent and Distributed Progrmming,* Prentice Hall, 1990.

[Bershad, *et.al.* 88]
Brian Bershad, E. Lazowska, H. Levy. *PRESTO : A System for Object-Oriented Paralell Programming.* Software-Practice and Experience, Vol. 18(8), August 1988.

[Bobrow y Stefik 86]
D. Bobrow y M. Stefik. *Perspectives on Artificial Intelligence Programming,* 'Science', Vol. 231, February 1986.

[Booch 94]
Grady Booch. *Object-Oriented Analysis and Design with Applications,* 2da. Edición, The Benjamin/Cummings Publishing Company Inc., 1994.

[Caromel 93]
Denis Caromel. *Toward a method of Object-Oriented Concurrent Programming,* Communications of the ACM, Vol. 36, No. 9, September 1993.

[Cardelli y Weger 85]
L. Cardelli y P. Wegner. *On undestanding Types, Data Abstraction and Polymorphism*, ACM Computing Surveys, Vol. 17 (4), December 1985.

[Faison 93]
Ted Faison. *BORLAND C++ 3.1, Programación Orientada a Objetos. Guía del Programador Avanzado para el Desarrollo de Aplicaciones Orientadas a Objetos*, SAMS, Prentice Hall, 1993.

[Franky 88]
M. C. Franky. *JOYCE+. Modelo, Lenguaje y Metodología de Programación de Sistemas Distribuidos sobre Redes de Comunicación*, XIV Conferencia Latinoamericana de Informática, 17 avas. Jornadas Argentinas de Informática e Investigación Operativa, Bs. Aires, Septiembre 1988.

[Greiff 93:a]
Warren R. Greiff. *El ocaso del Sol a través de una ventana*, Soluciones Avanzadas, No. 5, Méx. D.F., Septiembre-Octubre de 1993.

[Greiff 93:b]
Warren R. Greiff. *Paradigma vs. metodología : El caso de la programación Orientada a Objetos (Parte I)*, Soluciones Avanzadas, No. 6, Méx. D.F., Noviembre-Diciembre de 1993.

[Greiff 94]
Warren R. Greiff. *Paradigma vs. metodología : El caso de la programación Orientada a Objetos (Parte II)*, Soluciones Avanzadas, No. 7, Méx. D.F., Enero-Febrero de 1993.

[Hansen 78]
P. Brinch Hansen. *Distributed Processes : A Concurrent Progrmming Concept*, Communications of the ACM, 21(11), 1978.

[Hekmatpour 92]
Sharam Hekmatpour. C++. Guía para programadores en C, Prentice Hall, 1992.

[Hoare 78]
C.A.R. Hoare. *Communicating Secuential Processes*, Communications of the ACM, 21(8), 1978.

[Jézéquel 93]
J. M. Jézéquel. *Transparent parallelisation trough reuse : between a compiler and a library approach*, Memorias ECOOP, Alemania, 1993.

[Oktaba 87]
Hanna Oktaba. *Programación Concurrente* (primera y segunda parte), publicaciones del IIMAS - UNAM, serie azul, 1987.

[Oktaba 93:a]
Hanna Oktaba. *Análisis y diseño Orientado a Objetos : Introducción al método de Booch*, Soluciones Avanzadas, No. 6, Méx. D.F., Noviembre-Diciembre de 1993.

[Oktaba 93:b]
Hanna Oktaba. *Análisis y diseño Orientado a Objetos*, Soluciones Avanzadas, No. 4, Méx. D.F., Julio-Agosto de 1993.

[Oktaba y Quintanilla 93]
Hanna Oktaba y Gloria Quintanilla. *Abstracción de datos en lenguajes Orientados a Objetos*, Soluciones Avanzadas, No. 5, Méx. D.F., Septiembre-Octubre de 1993.

[Olmedo 91]
José Oscar Olmedo Aguirre. *Programación Concurrente Orientada a Objetos*, Centro de Investigación y de Estudios Avanzados del IPN, Dpto. de Ing. Eléctrica, Sección de Computación serie amarilla, No. 119, Mayo 1992.

[Olmedo 93]
José Oscar Olmedo Aguirre. *El lenguaje C++ Concurrente. Sinopsis y ejemplos*, Centro de Investigación y de Estudios Avanzados del IPN, Dpto. de Ing. Eléctrica, Sección de Computación, serie verde, No. 40, Junio 1993.

[Pedregal 89]
Cristobal Pedregal. *Issues in Object-Oriented Concurrency*, 'draft' de su Trabajo de grado en la Escuela Superior Latinoamericana de Informática (ESLAI), bajo la dirección de Carlo Ghezzi y Norma Lijtmaer, Septiembre de 1989.

[Quintanilla y Silva 93]
Gloria Quintanilla Osorio y Sergio Silva Barradas. *Elementos de la Programación Orientada a Objetos*, Soluciones Avanzadas, No. 4, Méx. D.F., Julio-Agosto de 1993.

[Rational Rose 92]
Rational Rose for Windows. *Rational Rose for Windows User's Manual*, Rev. 1.3, December 1993.

[Rational Rose 93]
Rational Rose for Windows (Iseult White). *The Booch Method : A Case Study for Rational Rose*, Rev. 1.0, January 1993.

[Romero 93]
Manuel Romero Salcedo. *Desarrollo e Implantación de Nuevos Mecanismos para la Comunicación y Sincronización en OCCAM y TRANSPUTERS*, tesis de Maestría en Ciencias de la Computación, UNAM.

[Karaorman y Bruno 93]
Murat Karaorman y John Bruno. *Introducing concurrency to a sequential Language*, Communications of the ACM, Vol. 36, No. 9, September 1993.

[Kafura y Lee]
D.G. Kafura y K.H. Lee*Inheritance in Actor based concurrent Object-Oriented Languages*, Proceedings of ECOOP 1989, Cambridge University Press.

[Liskov 88]
B. Liskov. *Distributed progrmming in Argus*, Communications of the ACM, Vol. 31, No. 3, March 1988.

[Lohr 93]
Kalus-Peter Lohr. *Concurrency annotations for reusable software*, Communications of the ACM, Vol. 36, No. 9, September 1993.

[Matsuoka y Yonezawa 93]
Satoshi Matsuoka y Akinori Yonezawa. *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*, MIT 1993.

[Meseguer 93]
José Meseguer. Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming, Memorias de ECOOP, Alemania, 1993.

[Meyer 88]
Bertrand Meyer. *Object - Oriented Software Construction*, Prentice Hall, 1988.

[Meyer 93]
Bertrand Meyer. *Systematic Concurrent Object-Oriented Programming*, Communications of the ACM, Vol. 36, No. 9, September 1993.

[Nierstrasz 87]
Oscar Nierstrasz. *Active Object in Hybrid*. Proceedings OOPSLA' 87, ACM SIGPLAN Notices, Col. 22, No. 12.

[Nierstrasz 93]
Oscar Nierstrasz. *Composing Active Objects*, MIT 1993.

[Saleh y Gautron]
Hayssan Saleh y Philippe Gautron. *Language Implementation Feedback on Design*, Rank Xerox France & Univ. ParisVI, Université Paris VI- LITP, [saleh,gautron]@rxf.ibp.fr, 1989 (ó mayor).

[Sánchez]
Víctor Germán Sánchez Arias. *Arquitectura heterogénea para un Sistema Distribuido*, Dpto. de Ciencias de la Computación, IIMAS, UNAM, 1990 (ó mayor).

[Stroustrup 93]
Bjarne Stroustrup. *The C++ Programming Language*, Addison-Wesley Publishing Company, 1993.

[Tanembaum 92]
Andrew S. Tanembaum. *Modern Operating Systems*, Prentice Hall, 1992.

[Wiener y Sincovec 90]
Richard Wiener y Richard Sincovec. *Programación en Ada*, Ed. Limusa Noriega, México, 1990.

[Yonezawa 87]
A. Yonezawa. *Modelling and programming in an Object-Oriented Concurrent Language ABCL/1*, Object-Oriented Concurrent Programming MIT Press, Cambridge, Mass., 1987.