Programación concurrente

Chapter · February 2008

CITATIONS
0

READS
535

2 authors:

Josep Jorba Esteve
Universitat Oberta de Catalunya
70 PUBLICATIONS **355** CITATIONS
SEE PROFILE

Remo Suppi
Autonomous University of Barcelona
87 PUBLICATIONS **231** CITATIONS
SEE PROFILE

Programación concurrente

Josep Jorba Esteve
Remo Suppi Boldrito

P07/M2106/02841

UOC
Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

Como consecuencia del rápido desarrollo de Internet, la programación distribuida está haciéndose rápidamente popular día a día. Internet provee de los primeros mecanismos básicos para una infraestructura global en las aplicaciones distribuidas, un espacio de nombres global (basado en las URL) y protocolos de comunicación globales (TCP/IP). La mayoría de las plataformas de sistemas distribuidos toman esta base para poder implementar diferentes modelos de aplicaciones y de sus usos.

Por otro lado, cuando generalizamos hacia el concepto de concurrencia, como habilidad de ejecutar múltiples actividades en paralelo o simultáneas, introducimos diferentes tipos de programación, como la paralela, la distribuida y la de memoria compartida. El conjunto de las cuales (lo denominaremos *programación concurrente*) se aplica en mayor o menor medida a diferentes sistemas distribuidos según el ámbito de aplicación.

En cuanto examinamos la programación concurrente, nos damos cuenta de su importancia actual, y de la prospección de futuro que tiene, ya sea desde el desarrollo de Internet y sus aplicaciones, o desde el nuevo hardware de las CPU (por ejemplo, en la forma de *multicores*), que nos introducirá la programación concurrente como un elemento básico para todo desarrollador de aplicaciones.

Los ambientes donde se desarrolla la programación concurrente, ya sea hoy o en un futuro próximo, están incrementándose paulatinamente, desde las ya mencionadas CPU *multicore*, y las tarjetas gráficas con sus procesadores concurrentes (y en este sentido, el campo del software en la programación de videojuegos), a las redes inalámbricas, y/o redes de sensores trabajando cooperativamente, pasando a las aplicaciones a nivel Internet, con los sistemas de compartición de información basados en mecanismos *peer-to-peer* (P2P).

También son aplicables a los diferentes ámbitos de aplicación, ya sean científicos (principalmente con computación en cluster, *grid*, o mediante hardware de máquinas paralelas o supercomputación) o en ambientes empresariales mediante diferentes arquitecturas por capas, y/o basadas en componentes o invocación remota de objetos, mediante las diferentes arquitecturas software empresariales.

La evolución de los sistemas distribuidos en los diferentes modelos arquitecturales ha creado un gran conjunto de posibilidades para la programación concurrente, que ha hecho surgir un gran número de paradigmas de programación.

Entendiendo estos paradigmas como clases de algoritmos que nos permiten solucionar diferentes problemas pero disponiendo de una misma estructura de control o concepción base.

Cada uno de ellos está mejor o peor adaptado a los diferentes ambientes distribuidos y/o paralelos, o más o menos especializado en arquitecturas de sistema concretas.

En este capítulo pretendemos observar los diferentes paradigmas, así como tener los conceptos teóricos y prácticos de los diferentes ambientes de programación para los sistemas concurrentes (distribuidos y/o paralelos), así como diferentes consideraciones de prestaciones a tener en cuenta en la implementación de las soluciones según los modelos arquitecturales, de interacción (comunicaciones), o de paradigmas de programación utilizados.

Un objetivo en especial es examinar las diferentes técnicas utilizadas en los diferentes ambientes distribuidos, ya sean computaciones de tipo distribuida, paralela, o *grid*. Daremos detalles, pero no entraremos en el debate, que no dispone de amplio consenso en la comunidad científica, sobre las diferencias de cada tipo de computación, distribuida, paralela, *grid*, o de memoria compartida (*shared memory*). Ya que hoy en día cada vez se difuminan más las diferencias, y en muchos casos la construcción de un sistema distribuido (y/o paralelo) conlleva el uso de uno o más modelos de computación, llevándonos a modelos híbridos dependiendo del ámbito de la aplicación.

En este sentido, consideramos la programación de tipo *multithread*, paralela y distribuida, como el ámbito de estudio del presente modulo, para proporcionar bases para el diseño e implementación de las aplicaciones dentro del ámbito de los sistemas distribuidos.

Objetivos

Los objetivos que tiene que conseguir el estudiante en este modulo didáctico son los siguientes:

1. Conocer los conceptos básicos de la programación concurrente, distribuida y paralela.

2. Conocer diferentes clasificaciones arquitecturales de los sistemas distribuidos.

3. Conocer los modelos de interacción en los sistemas distribuidos.

4. Conocer los diferentes paradigmas de programación utilizados en los sistemas distribuidos.

5. Conocer detalles de algunas de las técnicas de programación.

6. Conocer cómo adecuar cada paradigma de programación al uso o tipo de aplicación deseado.

7. Conocer cómo aplicar consideraciones de prestaciones para aumentar el rendimiento de los sistemas distribuidos.

1. Conceptos y definiciones

La programación a la que estamos más acostumbrados, la secuencial, estuvo en sus inicios fuertemente influenciada por las arquitecturas de único procesador, en las cuales disponíamos como características base:

- de un único procesador (CPU),
- de los programas y los datos que están almacenados en memoria RAM, y
- de los procesadores, que se dedican básicamente a obtener un determinado flujo de instrucciones desde la RAM. Ejecutando una instrucción por unidad de tiempo

En este sentido los programas secuenciales son totalmente ordenados, éstos nos indican en qué orden serán ejecutadas las instrucciones.

Y una particularidad importante, los programas secuenciales son deterministas: mediante una misma secuencia de datos de entrada, se ejecutará la misma secuencia de instrucciones y se producirá el mismo resultado (salvo errores de ejecución causados por causas externas). Esta afirmación no es cierta para la programación concurrente.

La programación concurrente nos permite desarrollar aplicaciones que pueden ejecutar múltiples actividades de forma paralela o simultánea.

La programación concurrente es necesaria por varias razones:

- Ganancia de procesamiento, ya sea en hardware multiprocesador o bien en un conjunto dado de computadoras. Mediante la obtención de ganancias en recursos, en capacidad de cómputo, memoria, recursos de almacenamiento, etc.

- Incremento del *throughput* de las aplicaciones (solapamiento E/S con cómputo).

- Incrementar la respuesta de las aplicaciones hacia el usuario, poder atender estas peticiones frente al cómputo simultáneo. Permitiendo a los usuarios y computadores, mediante las capacidades extras, colaborar en la solución de un problema.

- Es una estructura más apropiada para programas que controlan múltiples actividades y gestionan múltiples eventos, con el objetivo de capturar la estructura lógica de un problema.

- Da un soporte específico a los sistemas distribuidos (como analizaremos más adelante). En particular como posibilidad de enlazar y hacer trabajar de forma conjunta a dispositivos físicamente independientes.

Otro punto con cierta controversia, ya que no hay acuerdo específico, son las diferencias entre los términos de *programación concurrente, distribuida, paralela*, y *multithread*. Todos ellos son usados en mayor o menor medida en la construcción de sistemas distribuidos y/o paralelos. En este material usaremos la palabra *programación concurrente* como marco global a los diferentes tipos de programación distribuida, paralela y *multithread*, tendencia que viene siendo habitual en la literatura del campo distribuido.

Aun así, hay una serie de diferencias que hay que puntualizar:

- Concurrencia frente a paralelismo: en paralelismo hay procesamiento de cómputo simultáneo físicamente, mientras en concurrencia el cómputo es simultáneo lógicamente, ya que el paralelismo implica la existencia de múltiples elementos de procesamiento (ya sean CPU, o computadores enteros) con funcionamiento independiente; mientras que la concurrencia no implica necesariamente que existan múltiples elementos de procesamiento, ya que puede ser simulada mediante las capacidades de multiprocesamiento del sistema operativo sobre la CPU. Un caso típico es la programación *multithread,* la existencia de múltiples hilos de ejecución en una misma aplicación compartiendo memoria, pudiendo ser esta ejecución paralela si se ejecuta en una máquina multiprocesador, o simplemente concurrente, lógicamente si se ejecuta en un solo procesador. Aun existiendo estas diferencias, en la práctica las mismas técnicas son aplicables a ambos tipos de sistemas.

- La palabra *paralelismo* suele asociarse con connotaciones de altas prestaciones en cómputo científico en una determinada plataforma hardware, habitualmente en supercomputadores, aunque cada vez se desplaza más a plataformas basadas en *clusters* locales (computadoras enlazadas por redes de propósito general, o bien redes de alta velocidad optimizadas para procesamiento paralelo). Típicamente, estas computadoras están físicamente en un mismo armario o sala de cómputo.

- La palabra *distribuido* se ha referido típicamente a aplicaciones, ejecutándose en múltiples computadoras (normalmente se implica además heterogeneidad de recursos), que no tenían por qué estar físicamente en un mismo espacio. Este término también relaja su significado debido a la extensión de los modelos paralelos, a ambientes como *multicluster* (múltiples *clusters* enlazados), o *grid*, que, aun siendo generalmente considerados como paralelos, se acercan más a los sistemas distribuidos.

- En general estos términos son ampliamente difundidos, y usados en diferentes comunidades con significados diferentes, aunque las distinciones tienden a

desaparecer, en especial cuando se consideran cada vez sistemas más potentes de multiprocesamiento en sobremesa, y la fusión de sistemas paralelos y distribuidos aumenta con entornos como *grid*. Llegando en la mayoría de los casos actuales, en general, a sistemas híbridos donde se integran los tres tipos de programación (o de sistemas) para resolver un problema determinado.

Podemos asimismo partir de una definición de sistema distribuido de computación como:

> Una colección de elementos de cómputo autónomos, ejecutándose en uno o más computadores, enlazados por una red de interconexión, y soportados por diferentes tipos de comunicaciones que permiten observar la colección como un sistema integrado.

La mayoría de sistemas distribuidos operan sobre redes de computadoras, pero también puede construirse un sistema distribuido que tenga sus elementos funcionando en un computador simple de tipo multitarea.

Como hemos mencionado, además de algunos esquemas clásicos de sistema distribuido basados únicamente en redes, también pueden incluirse a día de hoy los computadores paralelos, y especialmente los servidores en *cluster*. Además de otros ambientes como las redes *wireless*, y las redes de sensores. Por otro lado, como vimos previamente a este módulo, la computación *grid* abarca coordinación de recursos que no están sujetos a control centralizado mediante el uso de protocolos e interfaces abiertas, de propósito general, para proporcionar diferentes servicios.

Uno de los parámetros más interesantes a la hora de enfocar la programación concurrente sobre un determinado sistema distribuido es conocer los diferentes estilos arquitectónicos en los que se basa, así como su modelo de interacción y organización interna de los componentes. Cuestiones que iremos desarrollando en las siguientes secciones del módulo.

En esta primera sección del módulo, examinaremos las formas en que se presentan las plataformas de programación concurrente, así como casos específicos de uso de la programación concurrente en el ámbito científico. Finalmente, observaremos qué soporte nos ofrece para la construcción, y el tratamiento de las problemáticas de los sistemas distribuidos.

1.1. Ambientes de computación concurrente

En la programación concurrente, los lenguajes suelen usar construcciones específicas para la concurrencia. Éstas pueden acarrear diferentes soportes a: *multithread*, soporte para computación distribuida, paso de mensajes, uso de recursos compartidos, etc.

En el caso de la programación concurrente, son usadas diferentes aproximaciones para el desarrollo del entorno de programación:

- API de desarrollo: normalmente, como serie de librerías añadidas a un lenguaje específico (en principio sin soporte concurrente). Estas librerías dan el soporte necesario al lenguaje para la implementación de diferentes paradigmas de programación.

  En algunas ocasiones, las librerías pueden traer alguna serie de servicios adicionales o utilidades que permiten la gestión de los sistemas distribuidos y/o paralelos creados, o ciertas modificaciones en los compiladores para el soporte necesario (como TBB, OpenMP, HPF, UPC). También podrían citarse las librerías de *sockets* para comunicaciones básicas TCP/IP, o bien las API para diferentes lenguajes para el soporte del paradigma de servicios web. En algunos casos de última generación, las API han sido integradas en la especificación del lenguaje, casos como Java y C# tienen soporte *multithread* nativo, así como paso de mensajes, e invocación remota de objetos, entre otras.

Bibliotecas añadidas a un lenguaje específico

Un ejemplo típico son las librerías de *threads* como POSIX (*pthreads*), para el soporte añadido de tipo *multithreading*. O APIs de paso de mensajes como MPI o PVM.

- *Middleware*: es una capa software distribuida que se sitúa sobre el sistema operativo (de red) y antes de la capa de aplicación, abstrayendo la heterogeneidad del entorno. Normalmente, provee de un entorno distribuido integrado con el objetivo de simplificar la tarea de programación y gestión de las aplicaciones distribuidas, generalmente se proporcionan una serie de servicios de valor añadido, como servicios de nombres, transacciones, etc., como mecanismos para facilitar el desarrollo de aplicaciones distribuidas. El objetivo principal es la integración e interoperabilidad de las aplicaciones y los servicios ejecutándose en plataformas heterogéneas y/o dispositivos de comunicaciones.

  Como ejemplos basados en diferentes paradigmas, podemos citar casos como CORBA, DCOM y RMI, que ofrecen modelos de programación de alto nivel. O en otros casos como SOAP (en web services) puede considerarse otro *middleware* (basado en XML), que permite a las aplicaciones intercambiar datos estructurados y tipados en la web. Otros ejemplos de diversos modelos, podrían ser Jini, JavaSpaces, Enterprise JavaBeans (EJBs), y los MOM (*message oriented middleware*).

- Lenguajes específicos: en algunos casos, sobre todo en diferentes trabajos de investigación, se han desarrollado lenguajes de programación en los que la concurrencia juega un rol importante, o han sido pensados directamente para soportar tareas concurrentes.

Ejemplos de lenguajes específicos

Podemos citar algunos ejemplos más conocidos como: ADA, Eiffel, Erlang Limbo, Linda, Occam.

1.2. Cómputo científico

La constante evolución del cómputo de altas prestaciones ha permitido que su uso se extienda a un amplio rango de campos de aplicación, tanto científicos

como comerciales. El constante avance en el hardware disponible (unido a la reducción en su coste) y la mejora de los paradigmas de programación han permitido disponer de entornos muy flexibles para atacar algunos de los grandes problemas de cómputo antes inabordables.

La computación paralela y/o distribuida de altas prestaciones comporta también unas implicaciones económicas y sociales importantes. Sus avances se reflejan a menudo en la competición industrial para producir mejores productos, y en algunos campos científicos como biología, química y medicina, por el bien social común en la mejor comprensión de mecanismos de enfermedades, drogas y medicamentos. También se ha iniciado el camino para grandes hitos futuros, como la exploración del conocimiento de los mecanismos del ADN humano, el análisis de la estructura de proteínas, y otros campos como la predicción de desastres naturales, terremotos, inundaciones, etc. La situación actual y las posibles mejoras en el cómputo de altas prestaciones nos abren una nueva serie de caminos para la ciencia, economía, ciencias sociales, etc.

Al campo ya clásico, en esta aproximación, de la supercomputación, se le ha añadido (dentro del mundo científico) en las últimas décadas, la computación distribuida, que ha puesto a disposición de un gran número de instituciones académicas y comerciales las capacidades de cómputo de altas prestaciones. Y en particular, las soluciones de cómputo basadas en *clusters* de ordenadores personales o servidores optimizados, ampliamente disponibles por ser soluciones de bajo coste.

En la mayoría de las ocasiones, estos avances de computación se han debido únicamente al avance en la tecnología del hardware, quedando relegado a un segundo término el software de sistemas, ya sea el propio sistema operativo o los paradigmas de programación usados.

Figura 1. Un supercomputador basado en *cluster* con 10.240 procesadores

En la última década, la aparición de implementaciones de los paradigmas de paso de mensajes para la programación de altas prestaciones, como los entornos de PVM y MPI, ha permitido incorporar esta programación a toda la gama de entornos hardware disponibles, desde las soluciones basadas en supercomputación hasta las más modestas basadas en *clusters* de variado tamaño (en número y capacidad de las máquinas que forman sus nodos). También recientemente, la disposición de máquinas de sobremesa con *multicore*, ha traído al escenario los paradigmas de programación para memoria compartida, en especial los modelos de *multithreading* (multihilo), y diversas implementaciones como los *threads* POSIX, Intel TBB o variaciones de lenguajes específicos como OpenMP.

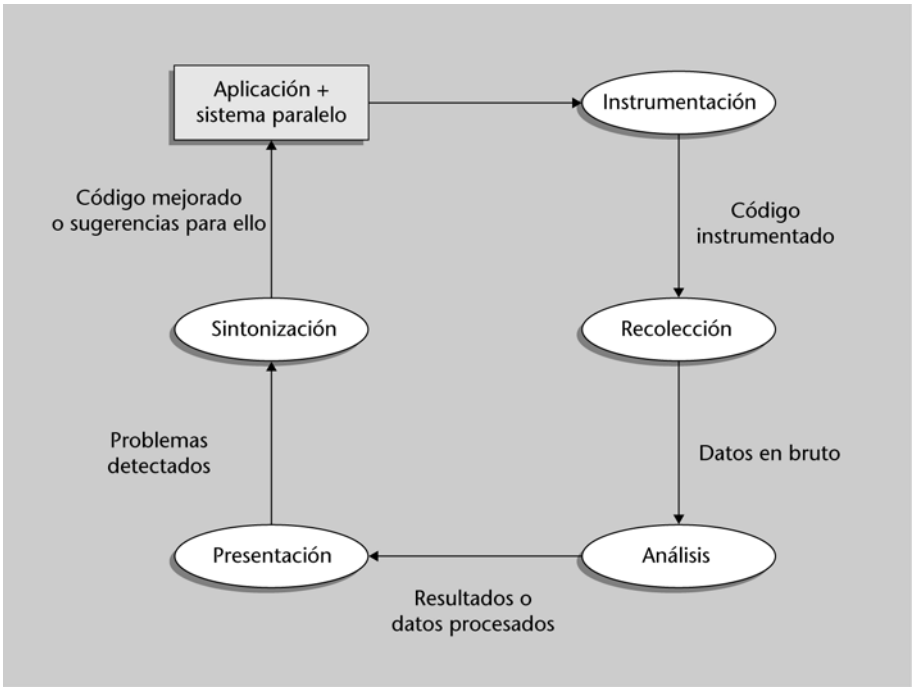
Nota

PVM y MPI son ejemplos de API basados en paradigmas de pago de mensajes. Los *thread* POSIX, Intel TBB, y OpenMP son casos de paradigma de memoria compartida.

Estas agrupaciones de paradigmas, tanto por paso de mensajes como de memoria compartida, han permitido atacar la resolución de diversos problemas que se implementan mejor en uno o en otro método, o con aproximaciones híbridas mediante *clusters* con nodos distribuidos, donde éstos son multiprocesador o *multicore*.

Pero tanto en un caso como en otro, hay que tener en cuenta que, para obtener las capacidades de cómputo que esperamos de estos entornos, hay que asegurar que las aplicaciones han sido correctamente diseñadas y que sus prestaciones son satisfactorias. En el caso científico, esto implica, especialmente en el caso de altas prestaciones, que las tareas del desarrollador no acaban con aplicaciones libres de fallos funcionales, sino que es necesaria la realización de un análisis de prestaciones y la sintonización adecuada para alcanzar los índices esperados de rendimiento. Un ciclo de vida típico del desarrollo en este ámbito podría ser:

Figura 2



Ya sea como evolución lógica o debido a la necesidad de abordar problemas con mayores requisitos de cómputo, en los últimos tiempos se ha complementado la evolución en el hardware, con el análisis de prestaciones de los entornos software para intentar extraer de ellos los máximos picos de cómputo, o los mejores rendimientos de la relación cómputo/prestaciones presentes en los paradigmas de paso de mensajes, o la mejor eficiencia de *cores* o procesadores en paradigmas de memoria compartida, o bien para aumentar la escalabilidad hardware/software de la solución empleada.

Por desgracia, cumplir estos requisitos en las aplicaciones distribuidas/paralelas no es fácil. Las razones de estas limitaciones de las aplicaciones (y de sus desarrolladores) están en las complejas interacciones que las aplicaciones soportan con el sistema físico donde se ejecutan, el software de sistema (operativo y librerías), las interfaces de programación usadas (cada vez de más alto nivel) y los algoritmos implementados.

Comprender todas estas interacciones y cómo mejorarlas es crucial para optimizar las prestaciones que podamos obtener de las aplicaciones, alcanzando

una mejor utilización y productividad de los recursos hardware disponibles, para maximizar las prestaciones de los sistemas disponibles.

1.3. Soporte de la programación concurrente a los sistemas distribuidos

En general, la programación concurrente es usada en la computación distribuida para dar solución a los diferentes problemas que aparecen en la concepción de sistemas distribuidos de cómputo:

- **Heterogeneidad**: el acceso a servicios, o la ejecución de aplicaciones, comporta un gran conjunto de heterogeneidad en computadores y redes. Aplicándose desde las propias redes al hardware del computador, a los sistemas operativos y a los lenguajes de programación que pueden encontrarse en el sistema final.

- **Tolerancia a fallos**: como habilidad de recuperar el sistema desde fallos de sus componentes sin perder la estabilidad del sistema.

- **Alta disponibilidad**: reducción al mínimo tiempo de parada de un sistema, cuando éste se está reconfigurando o ajustando, de manera que se minimice el tiempo que no está disponible de cara a sus clientes.

- **Continua disponibilidad**: capacidad de proporcionar servicio de forma ininterrumpida.

- **Recuperabilidad**: capacidad de recuperarse de los fallos reactivando los elementos del sistema, e incorporarlos de nuevo después de los fallos.

- **Consistencia**: habilidad para coordinar múltiples acciones en múltiples elementos, en presencia de concurrencia y fallos. En cierta forma, puede verse como comparación a un sistema no distribuido, y cómo conseguir comportarse como éste.

- **Escalabilidad**: habilidad para que el sistema se siga comportando correctamente a pesar de que diferentes aspectos del sistema crezcan en tamaño u otras características. Podemos observar la escalabilidad desde diferentes factores, aumento de capacidad de cómputo, de capacidad de red, de número de clientes del sistema, etc. Lo que nos obliga a tomar consideraciones de cómo afectarán al sistema estos cambios.

- **Seguridad**: capacidad de proteger los datos, servicios y recursos frente a ataques de usuarios malintencionados.

- **Privacidad**: protección de la identidad y localización de los usuarios del sistema, así como de los contenidos con datos sensibles de usos no autorizados.

- **Prestaciones predecibles**: garantizar que el sistema obtiene unas prestaciones acordes con las expectativas, como por ejemplo la ratio de datos producida o tratada, latencias controladas en caminos críticos, peticiones procesadas por unidad de tiempo, u otras.

- **Actualidad de los datos**: en especial en sistemas con restricciones temporales, asegurar que las acciones sean tomadas en los límites temporales pertinentes, y que sean realizadas con la sincronización temporal suficiente entre los elementos del sistema distribuido.

- **Abertura** (*openness*): es la característica que nos permite extender un sistema o reimplementarlo en diferentes formas. En el caso distribuido, podemos verlo como la facilidad con la que nuevos recursos pueden ser añadidos y puestos a disposición. En particular un sistema abierto se caracterizará por disponer de interfaces conocidas y publicadas, con mecanismos comunes conocidos de intercomunicación normalmente basados en estándares reconocidos. Estos sistemas abiertos pueden ser construidos desde hardware y software heterogéneo proveniente de diferentes fuentes. Pero debe cumplirse el seguimiento de las serie de estándares para que cada elemento pueda ser testado y verificado de forma que se compruebe su adecuación al sistema y correcto funcionamiento.

2. Clasificaciones arquitecturales

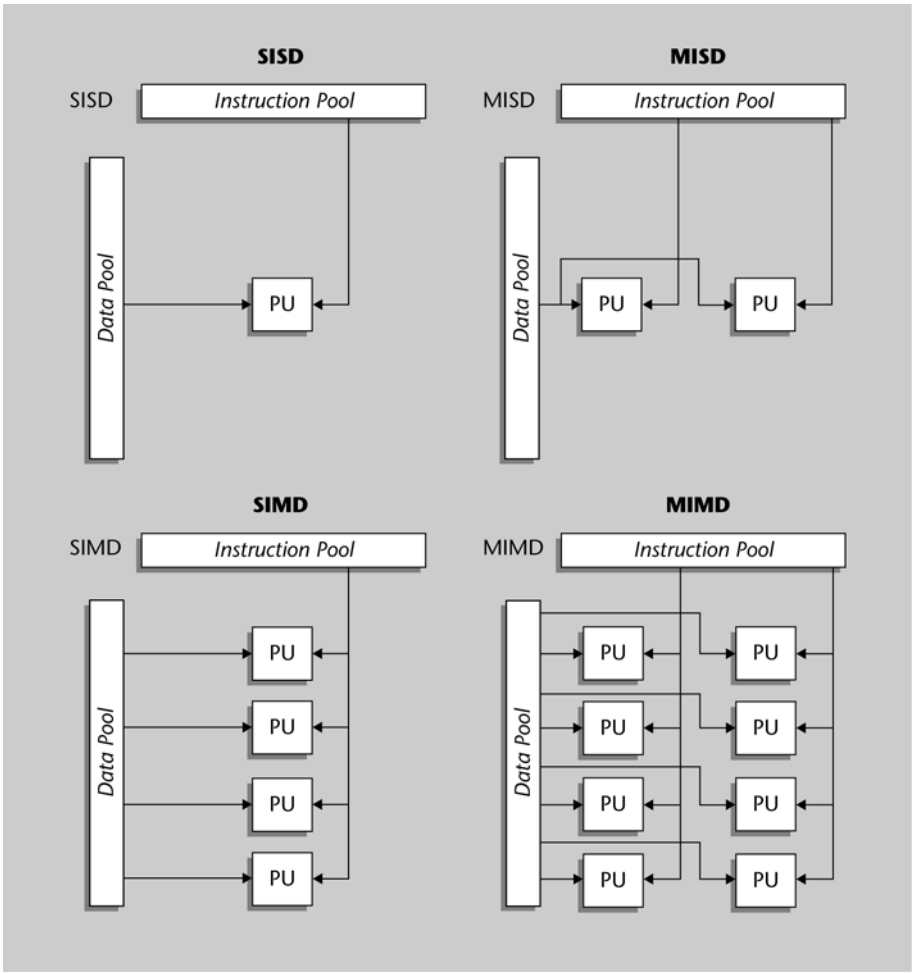
Para la concepción de sistemas concurrentes y sus paradigmas de programación, hay que tener en cuenta que tanto en el caso distribuido, como en el paralelo los sistemas disponen de múltiples procesadores, que son capaces de trabajar conjuntamente en una o varias tareas para resolver un problema computacional.

Existen muchas formas diferentes de construir y clasificar en diferentes taxonomías los sistemas distribuidos y paralelos. Presentaremos varias clasificaciones basadas en diferentes conceptos.

2.1. Taxonomía de Flynn

Ésta es una clasificación de sistemas arquitecturales muy usada en paralelismo y en cómputo científico, que nos sirve para identificar tipos de aplicaciones y sistemas en función de los flujos de instrucciones (ya sean *threads*, procesos o tareas) y los recursos disponibles para ejecutarlos.

Figura 3



Taxonomía de Flynn

En función de los conjuntos de instrucciones y datos, y frente a la arquitectura secuencial que denominaríamos SISD (*single instruction single data*), podemos clasificar las diferentes arquitecturas paralelas (o distribuidas) en diferentes grupos: SIMD (*single instruction multiple data*), MISD (*multiple instruction single data*) y MIMD (*multiple instruction multiple data*), con algunas variaciones como la SPMD (*single program multiple data*).

- SIMD: un solo flujo de instrucciones es aplicado a múltiples conjuntos de datos de forma concurrente. En una arquitectura SIMD, unos procesos homogéneos (con el mismo código) sincrónicamente ejecutan la misma instrucción sobre sus datos, o bien la misma operación se aplica sobre unos vectores de tamaño fijo o variable. El modelo es válido para procesamientos matriciales y vectoriales, siendo las máquinas paralelas con procesadores vectoriales un ejemplo de esta categoría.

- MISD: el mismo conjunto de datos se trata de forma diferente por los procesadores. Son útiles en casos donde sobre el mismo conjunto de datos se deban realizar muchas operaciones diferentes. En la práctica no se han construido máquinas de este tipo por las dificultades en su concepción.

- MIMD: paralelismo funcional y/o de datos. No sólo distribuimos datos, sino también las tareas a realizar entre los diferentes procesadores/nodos. Varios flujos (posiblemente diferentes) de ejecución son aplicados a diferentes conjuntos de datos. Esta categoría no es muy concreta, ya que existe una gran variedad de arquitecturas posibles con estas características, incluyendo máquinas con varios procesadores vectoriales o sistemas de centenares de procesadores o bien con unos pocos.

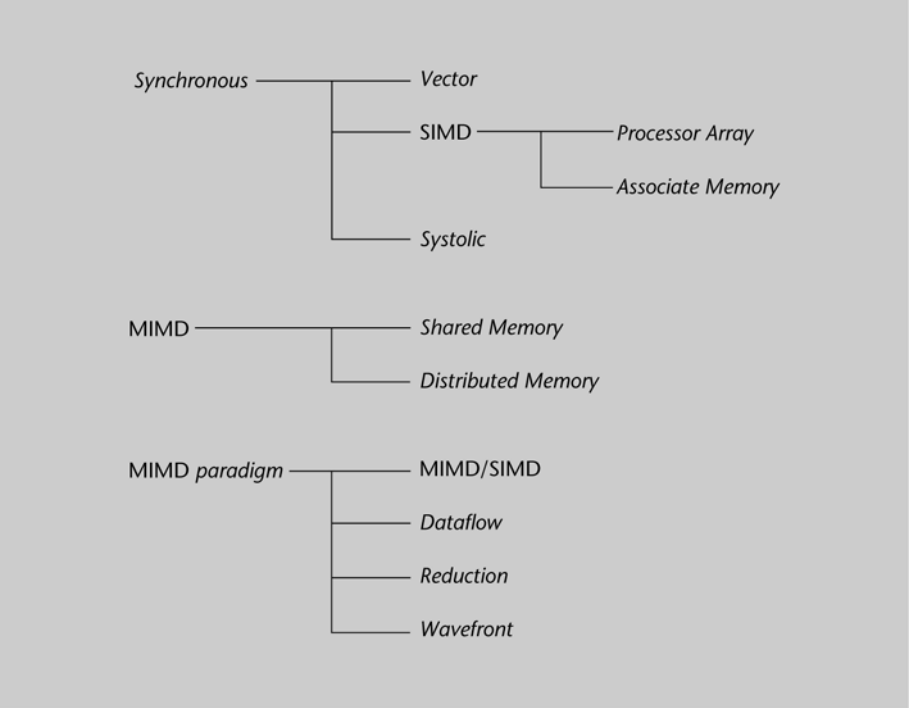
- SPMD: en paralelismo de datos, utilizamos mismo código con distribución de datos. Hacemos varias instancias de las mismas tareas, cada uno ejecutando el código de forma independiente. SPMD puede verse como una extensión de SIMD o bien una restricción del MIMD. A veces suele tratarse más como un paradigma de programación, en el cual el mismo código es ejecutado por todos los procesadores (u nodos) del sistema, pero en la ejecución se pueden seguir diferentes caminos en los diferentes procesadores.

Esta clasificación (taxonomía de Flynn) de modelos arquitecturales se suele ampliar para incluir diversas categorías de ordenadores que no se ajustan totalmente a cada uno de estos modelos. Una clasificación extendida de Flynn podría ser la siguiente:

En esta clasificación (ver figura 4), se ha extendido la clásica para incluir desarrollos de máquinas con diseños arquitectónicos concretos (más relacionados con máquinas paralelas). En particular, comentar la inclusión de las computadoras vectoriales en la categoría SIMD. Estas máquinas incorporan una CPU clásica (bajo el modelo von Neumann), pero con vectores como tipo de datos

primitivo. Tenemos instrucciones del repertorio que permiten operar sobre todas las componentes del vector. Estas máquinas con uno o más procesadores vectoriales (en algunos casos MIMD), son las que clásicamente se han conocido con la denominación de *supercomputadores* (en particular las computadoras vectoriales del fabricante Cray), aunque algunas de las características de procesamiento vectorial comienzan a estar disponibles en las recientes arquitecturas de procesadores de sobremesa (en especial para proceso multimedia).

Figura 4



2.2. Por control y comunicación

Otra extensión de las clasificaciones anteriores de los modelos arquitecturales es la basada en los mecanismos de control y la organización del espacio de direcciones en el caso de las arquitecturas MIMD, las de más amplia repercusión hoy en día (y que se ajusta bien a diferentes sistemas distribuidos):

- MIMD con memoria compartida (*shared memory,* MIMD), o también denominados multiprocesadores: En este modelo, los procesadores comparten el acceso a una memoria común. Existe un único espacio de direcciones de memoria para todo el sistema, accesible a todos los procesadores. Los procesadores se comunican a través de esta memoria compartida. Los desarrolladores no tienen que preocuparse de la posición actual de almacenamiento de los datos, y todos los procesadores tienen el mismo espacio de acceso a los mismos. En particular, cabe destacar la denominación comercial de *sistemas SMP* (*symetric multiprocessing*), refiriéndose a la combinación de MIMD y memoria (físicamente) compartida. Sus principales limitaciones son el número de procesadores dependiendo de la red de interconexión,

siendo habitual sistemas entre 2-4 a 16 procesadores, incluidos en una misma máquina. Aunque también existe la opción, por parte de varios fabricantes, de unir algunos de estos sistemas (formando un sistema híbrido), en lo que se conoce como *SPP* (*scalable parallel processing*) o *CLUMP* (*clusters of multiprocessors*), para formar un sistema mayor. Como resumen, podemos decir que estas máquinas (SMP y SPP) ofrecen flexibilidad y fácil programación (por el esquema de memoria compartida), a costa de complejidad adicional en el hardware.

- Sistemas MIMD con memoria distribuida (*distributed memory,* MIMD), o también llamados multicomputadores: En este modelo, cada procesador ejecuta un conjunto separado de instrucciones sobre sus propios datos locales. La memoria no centralizada está distribuida entre los procesadores (o nodos) del sistema, cada uno con su propia memoria local, en la que poseen su propio programa y los datos asociados. El espacio de direcciones de memoria no está compartido entre los procesadores. Una red de interconexión conecta los procesadores (y sus memorias locales), mediante enlaces (*links*) de comunicación, usados para el intercambio de mensajes entre los procesadores. Los procesadores intercambian datos entre sus memorias cuando se pide el valor de variables remotas. También podemos observar unas subdivisiones de este modelo MIMD de memoria distribuida, Por un lado:

  a) Una extensión natural de este modelo sería el uso de una red de computadoras (evidentemente con una latencia más grande que la que presenta una red de interconexión dedicada en una máquina paralela). En este caso, cada nodo de la red es en sí mismo una computadora completa, pudiendo incluso operar de forma autónoma del resto de nodos. Dichos nodos pueden, además, estar distribuidos geográficamente en localizaciones distintas. A estos sistemas se les suele denominar *clusters*, y los analizaremos con más detalle posteriormente.

  b) Comercialmente, es habitual denominar a los sistemas MIMD con memoria físicamente distribuida y red de interconexión dedicada, especialmente diseñada, *sistemas MPP* (*massive parallel processing*). En éstos, el número de procesadores puede variar desde unos pocos a miles de ellos. Normalmente, los procesadores (de un sistema MPP) están organizados formando una cierta topología (tanto física como lógica, como: anillo, árbol, malla, hipercubo, etc.), disponiendo de un red de interconexión entre ellos de baja latencia y gran ancho de banda.

  En esta última clasificación, es necesario tener en cuenta la diferenciación de la organización de la memoria, entre la organización física y la lógica. Podemos, por ejemplo, disponer de una organización física de memoria distribuida, pero lógicamente compartida. En el caso de los multiprocesadores, podríamos realizar una subdivisión más entre sistemas fuertemente o débilmente acoplados.

En un sistema fuertemente acoplado, el sistema ofrece un mismo tiempo de acceso a memoria para cada procesador. Este sistema puede ser implementado a través de un gran módulo único de memoria, o por un conjunto de módulos de memoria de forma que se pueda acceder a ellos en paralelo por los diferentes procesadores. El tiempo de acceso a memoria (a través de la red de interconexión común) se mantiene uniforme para todos los procesadores. Tenemos un acceso uniforme a memoria (UMA).

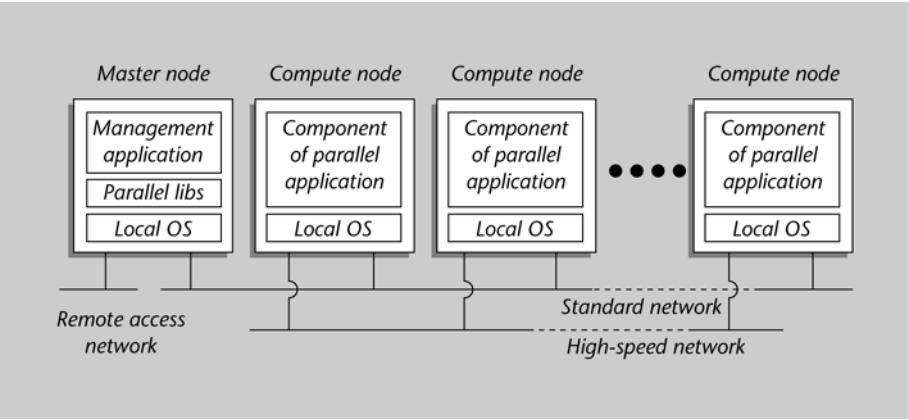
En un sistema ligeramente acoplado, el sistema de memoria está repartido entre los procesadores, disponiendo cada uno de su memoria local. Cada procesador puede acceder a su memoria local y a la de los otros procesadores, pero el tiempo de acceso a las memorias no locales es superior a la de la local, y no necesariamente igual en todas. Tenemos un acceso no uniforme a la memoria (NUMA).

Por último, analizamos en esta sección, como caso que merece especial atención, los *clusters*, que pueden verse como un tipo especial de sistema MIMD de memoria distribuida (DM-MIMD), los cuales tienen una amplia repercusión en el desarrollo de sistemas de alta disponibilidad y alta fiabilidad.

2.3. *Clusters* y *grid*

Los *clusters* consisten en una colección de ordenadores (no necesariamente homogéneos) conectados por red ya sea Gigabit u Ethernets de menor velocidad, Fiber Channel, ATM, u otras tecnologías de red para trabajar concurrentemente en tareas del mismo programa. Un *cluster* está controlado por una entidad administrativa simple (normalmente centralizada), que tiene el control completo sobre cada sistema final. Como modelo de arquitectura, son sistemas MIMD de memoria distribuida, aunque la interconexión puede no ser dedicada, y utilizar mecanismos de interconexión simples de red local, o incluso a veces no dedicados exclusivamente al conjunto de máquinas. Básicamente, es un modelo DM-MIMD, pero las comunicaciones entre procesadores son, habitualmente, varias órdenes de magnitud más lentas, que las que se producirían en una máquina paralela.

Figura 5



Ejemplo de configuración de un *cluster*

Esta combinación (homogénea o heterogénea) de máquinas podemos utilizarla como computador paralelo/distribuido, gracias a la existencia de paquetes software que permiten ver el conjunto de nodos disponibles como una máquina paralela virtual, ofreciendo una opción práctica, económica y popular hoy en día para aproximarse al cómputo paralelo. Algunas de las ventajas que podemos encontrar en estos sistemas son las siguientes:

- Cada máquina del *cluster* es en sí misma un sistema completo, utilizable para un amplio rango de aplicaciones de usuario. Pero estas aplicaciones normalmente no consumen de forma constante un tiempo apreciable de CPU. Una posibilidad es usar las máquinas del *cluster*, aprovechando estos tiempos (o porcentajes de tiempo) muertos para utilizarlos en las aplicaciones paralelas, sin que esto perjudique a los usuarios convencionales de las máquinas. Esto también nos permite diferenciar entre los *clusters* dedicados (o no), dependiendo de la existencia de usuarios (y por tanto de sus aplicaciones) conjuntamente con la ejecución de aplicaciones paralelas. Asimismo, también podemos diferenciar entre dedicación del *cluster* a una sola o varias aplicaciones paralelas simultáneas en ejecución.

- El aumento significativo de la existencia de los sistemas en red con un amplio mercado y su comercialización ha hecho que el hardware para estos sistemas sea habitual y económico, de manera que podemos montar una máquina paralela virtual a bajo coste. A diferencia de las máquinas SMP y otros superordenadores, es más fácil conseguir buenas relaciones de coste y prestaciones. De hecho, buena parte de los sistemas actuales de altas prestaciones son sistemas en *cluster*, con redes de interconexión más o menos dedicadas.

- Los podríamos situar como una alternativa intermedia entre los sistemas comerciales basados en SMP y MPP. Sus prestaciones van mejorando en la medida en que se desarrolla software específico para aprovechar las características de los sistemas en *cluster*, al tiempo que se introducen mejoras en las tecnologías de red local y de medio alcance.

- La computación en *cluster* es bastante escalable (en hardware). Es fácil construir sistemas con centenares o miles de máquinas. Por el contrario, las SMP suelen estar limitadas en número de procesadores. De hecho, en los *clusters*, en la mayor parte de las ocasiones, la capacidad se ve limitada por las tecnologías de red, debiendo ser las redes de interconexión las que puedan soportar el número de máquinas que queramos conectar, ya sea utilizando tecnologías de red generales (Ethernet, *fast* o *giga ethernet*, ATM, etc.) o bien redes de altas prestaciones dedicadas (Myrinet, SCI, etc.). Además, los entornos software para *cluster* ofrecen una gran escalabilidad desde cómputo en pequeños *clusters*, como los sistemas de paso de mensajes, hasta sistemas de cómputo conectados a través de Internet, como con los sistemas *grid*.

- Aunque no es habitual disponer de hardware tolerante a fallos (como es el caso de SMP y supercomputadores), normalmente se incorporan mecanismos software/hardware que invalidan (y/o sustituyen por otro) el recurso que ha fallado.

Pero también hay una serie de problemas relevantes a considerar:

- Con contadas excepciones, el hardware de red no está diseñado para el procesamiento paralelo. La latencia típica es muy alta, y el ancho de banda es relativamente bajo comparado con sistemas SMP o supercomputadores.

- El software suele diseñarse para máquinas personales. Tal es el caso del sistema operativo, y normalmente no ofrece posibilidades de control (y gestión) de *clusters*. En estos casos, es necesario incorporar una serie de capas de servicios *middleware* sobre el sistema operativo, para proporcionar sistemas en *cluster* que sean eficaces. En sistemas a mayor escala, es necesario incluir toda una serie de mecanismos de control y gestión adicionales, para el *scheduling* y monitorización de los sistemas, como es el caso en los sistemas *grid*. En general, esto supone una complejidad muy grande del software de sistema, que aumenta sensiblemente la complejidad total, y tiene una repercusión significativa sobre las prestaciones en estos sistemas.

3. Descomposición de problemas

La descomposición de los problemas a resolver plantea diferentes retos a nivel distribuido y paralelo, las aplicaciones distribuidas y/o paralelas consisten en una o más tareas que pueden comunicarse y cooperar para resolver un problema.

Por *descomposición* entendemos la división de las estructuras de datos en subestructuras que pueden distribuirse separadamente, o bien una técnica para dividir la computación en computaciones menores, que pueden ser ejecutadas separadamente.

Las estrategias más comúnmente usadas incluyen:

- Descomposición funcional: se rompe el cómputo en diferentes subcálculos, que pueden: a) realizarse de forma independiente; b) en fases separadas (implementándose en *pipeline*); c) con un determinado patrón jerárquico o de dependencias de principio o final entre ellos.

- Descomposición geométrica: el cálculo se descompone en secciones que corresponden a divisiones físicas o lógicas del sistema que se está modelando. Para conseguir un buen balanceo, estas secciones deben ser distribuidas de acuerdo a alguna regla regular o repartidas aleatoriamente. Normalmente, es necesario tener en cuenta cierta ratio de cómputo-comunicación, para realizar un balanceo más o menos uniforme.

- Descomposición iterativa: romper un cómputo en el cual una o más operaciones son repetidamente aplicadas a uno o más datos, ejecutando estas operaciones de forma simultánea sobre los datos. En una forma determinística, los datos a procesar son fijos, y las mismas operaciones se aplican a cada uno. En la forma especulativa, diferentes operaciones son aplicadas simultáneamente a la misma entrada hasta que alguna se complete.

En cuanto a la creación de las aplicaciones distribuidas o paralelas, basándose en las posibles descomposiciones, no hay una metodología claramente establecida ni fija, debido a la fuerte dependencia de las arquitecturas de las máquinas que se usen, y los paradigmas de programación usados en su implementación.

3.1. Una metodología básica

Una metodología simple de creación de aplicaciones paralelas y/o distribuidas podría ser la que estructura el proceso de diseño en cuatro etapas diferentes:

partición, comunicación, aglomeración y *mapping* (a veces a esta metodología se la denomina con el acrónimo PCAM). Las dos primeras etapas se enfocan en la concurrencia y la escalabilidad, y se pretende desarrollar algoritmos que primen estas características. En las dos últimas etapas, la atención se desplaza a la localidad y las prestaciones ofrecidas.

- Partición: el cómputo a realizar y los datos a operar son descompuestos en pequeñas tareas. El objetivo se centra en detectar oportunidades de ejecución concurrente. Para diseñar una partición, observamos los datos del problema, determinamos particiones de estos datos y, finalmente, se asocia el cómputo con los datos. A esta técnica se la denomina *descomposición del dominio*. Una aproximación alternativa consiste en la descomposición funcional, asignando diferentes cómputos o fases funcionales a las diferentes tareas. Las dos son técnicas complementarias que pueden ser aplicadas a diversos componentes o fases del problema, o bien aplicadas al mismo problema para obtener algoritmos distribuidos o paralelos alternativos.

- Comunicación: se determinan las comunicaciones necesarias (en forma de estructuras de datos necesarias, protocolos, y algoritmos), para coordinar la ejecución de las tareas.

- Aglomeración: las tareas y estructuras de comunicación de las dos primeras fases son analizadas respecto de las prestaciones deseadas y los costes de implementación. Si es necesario, las tareas son combinadas en tareas mayores, si con esto se consigue reducir los costes de comunicación y aumentar las prestaciones.

- *Mapping*: cada tarea es asignada a un procesador/nodo, de manera que se intentan satisfacer los objetivos de maximizar la utilización del procesador/nodo, y minimizar los costes de comunicación. El *mapping* puede especificarse de forma estática, o determinarlo en ejecución mediante métodos de balanceo de carga.

El resultado de esta metodología, dependiendo de los paradigmas y las arquitecturas físicas, puede ser una aplicación distribuida y/o paralela, con un *mapping* estático entre tareas y procesadores/nodos a la hora de iniciar la aplicación, o bien una aplicación que crea tareas (y las destruye) de forma dinámica, mediante técnicas de balanceo de carga. Alternativamente, también podemos utilizar estrategias de tipo SPMD que crea exactamente una tarea por procesador/nodo, asumiendo que la etapa de aglomeración ya combina las tareas de *mapping*.

Una de las decisiones más significativas en las aplicaciones distribuidas o paralelas, como resultado de las tareas anteriores, es precisamente el paradigma de programación que usaremos. Con cada uno de los paradigmas de programación, nos estamos refiriendo a una clase de algoritmos que tienen la misma

estructura de control, y que pueden ser implementados usando un modelo genérico de programación. Más adelante veremos diferentes paradigmas.

3.2. Modelos de descomposición

En las próximas secciones introduciremos los modelos más relevantes usados de programación distribuida/paralela. Para los modelos de programación, remarcamos cómo solucionan la distribución de código y la interconexión entre las unidades de ejecución y las tareas. Cualquiera de estos modelos de programación puede ser usado para implementar los paradigmas de programación. Pero las prestaciones de cada combinación resultante (paradigma en un determinado modelo) dependerán del modelo de ejecución subyacente (la combinación de hardware, red de interconexión y software de sistema disponibles).

3.2.1. Memoria compartida

En este modelo los programadores ven sus programas como una colección de procesos accediendo a variables locales y un conjunto de variables compartidas. Cada proceso accede a los datos compartidos mediante una lectura o escritura asíncrona. Por tanto, como más de un proceso puede realizar las operaciones de acceso a los mismos datos compartidos en el mismo tiempo, es necesario implementar mecanismos para resolver los problemas de exclusiones mutuas que se puedan plantear, mediante mecanismos de semáforos o bloqueos.

En este modelo, el programador ve la aplicación como una colección de tareas que normalmente son asignadas a *threads* de ejecución en forma asíncrona. Los *threads* poseen acceso al espacio compartido de memoria, con los mecanismos de control citados anteriormente. En cuanto a los mecanismos de programación utilizados, pueden usarse las implementaciones de *threads* en diferentes operativos, y los segmentos de memoria compartida, así como paralelismo implícito en algunos casos. En este último, se desarrollan aplicaciones secuenciales, donde se insertan directivas que permiten al compilador realizar distribuciones o paralelizaciones de código en diversas secciones.

OpenMP es una de las implementaciones más utilizadas para programar bajo este modelo en sistemas de tipo SMP (o SH-MIMD). OpenMP (*open specifications for multi processing*) define directivas y primitivas de librería para controlar la paralelización de bucles y otras secciones de un código en lenguajes como Fortran, C y C++. La ejecución se basa en la creación de *thread* principal, juntamente con la creación de *threads* esclavos cuando se entra en una sección paralela. Al liberar la sección, se reasume la ejecución secuencial. OpenMP, que normalmente se implementa a partir librerías de bajo nivel de *threads*.

En OpenMP se usa un modelo de ejecución paralela denominado *fork-join*, básicamente el *thread* principal, que comienza como único proceso, realiza en un momento determinado una operación *fork* para crear una región paralela (directivas PARALLEL, END PARALLEL) de un conjunto de *threads*, que acaba mediante una sincronización por una operación *join*, reasumiéndose el *thread* principal de forma secuencial, hasta la siguiente región paralela. Todo el paralelismo de OpenMP se hace explícito mediante el uso de directivas de compilación que están integradas en el código fuente (Fortran, o C/C++).

En caso de sistemas físicos paralelos de tipo híbrido, que soporten por ejemplo nodos SMP en *cluster* o de tipo SPP, suele combinarse la programación OpenMP con otros modelos como el de paso de mensajes.

3.2.2. Paralelismo de datos

El paralelismo de datos es un paradigma en el cual operaciones semejantes (o iguales) son realizadas sobre varios elementos de datos simultáneamente, por medio de la ejecución simultánea en múltiples procesadores.

Este modelo es aconsejable para las aplicaciones que realizan la misma operación sobre diferentes elementos de datos. La idea principal es explotar la concurrencia que deriva de que la aplicación realice las mismas operaciones sobre múltiples elementos de las estructuras de datos.

Un programa paralelo de datos consiste en una lista de las mismas operaciones a realizar en una estructura de datos. Entonces, cada operación en cada elemento de datos puede realizarse como una tarea independiente.

El paralelismo de datos es considerado como un paradigma de más alto nivel, en el cual, al programador no se le requiere que haga explícitas las estructuras de comunicación entre los elementos participantes. Éstas son normalmente derivadas por un compilador, que realiza la descomposición del dominio de datos a partir de indicaciones del programador sobre la partición de los datos. En este sentido, hemos de pensar que el programa posee una semántica de programa secuencial, y solamente debemos pensar en el entorno paralelo para la selección de la distribución de los datos teniendo en cuenta la ayuda del compilador y las posibles directivas que le podamos proporcionar para guiar el proceso.

El modelo de diseño de aplicaciones paralelas PCAM (mencionado previamente) sigue siendo aplicable, aunque en los lenguajes de paralelismo de datos, se realiza la primera fase directamente por medio de construcciones implícitas y otras explícitas proporcionadas por el usuario, de cara a obtener una granularidad fina de computación. Un punto relevante en esta fase es identificar (mediante el compilador y la ayuda proporcionada por el usuario) particiones con

un grado suficiente de concurrencia. En el caso del paralelismo de datos, las restantes fases del modelo de diseño PCAM pueden realizarse de forma práctica gracias a directivas de compilador de alto nivel, en lugar de pensar en términos de comunicaciones explicitas y operaciones de *mapping* a nodos de cómputo.

La conversión del programa, así descrito mediante estructuras de granularidad fina de cómputo y directivas a un ejecutable (típicamente de tipo SPMD), es un proceso automático que realiza el compilador de paralelismo de datos. Pero hay que tener en cuenta que el programador tiene que entender las características esenciales de este proceso, procurando escribir código eficiente y evitando construcciones ineficientes. Por ejemplo, una elección incorrecta de algunas directivas puede provocar desbalanceos de carga o comunicaciones innecesarias. Asimismo, el compilador puede fallar en los intentos de reconocer oportunidades para la ejecución concurrente. Generalmente, se puede esperar que un compilador para paralelismo de datos sea capaz de generar código razonablemente eficiente cuando la estructura de comunicaciones sea local y regular. Programas con estructuras irregulares o comunicaciones globales pueden tener problemáticas significativas en cuanto a las prestaciones obtenidas.

En cuanto a los lenguajes que soportan este modelo, podemos citar a Fortran 90, y su extensión High Performance Fortran, como el más ampliamente utilizado para implementar este tipo de paralelismo.

3.2.3. Paso de mensajes

Éste es uno de los modelos más ampliamente usado. En él los programadores organizan sus programas como una colección de tareas con variables locales privadas y la habilidad de enviar, y recibir datos entre tareas por medio del intercambio de mensajes. Definiéndose así por sus dos atributos básicos: Un espació de direcciones distribuido y soporte únicamente al paralelismo explícito.

Los mensajes pueden ser enviados vía red o usando memoria compartida si está disponible. Las comunicaciones entre dos tareas ocurren a dos bandas, donde los dos participantes tienen que invocar una operación. Podemos denominar a estas comunicaciones como operaciones cooperativas, ya que deben ser realizadas por cada proceso, el que tiene los datos y el proceso que quiere acceder a los datos. En algunas implementaciones, también pueden existir comunicaciones de tipo *one-sided*, si es sólo un proceso el que invoca la operación, colocando todos los parámetros necesarios y la sincronización se hace de forma implícita.

Como ventajas, el paso de mensajes permite un enlace con el hardware existente, ya que se corresponde bien con arquitecturas que tengan una serie de

procesadores conectados por una red de comunicaciones (ya sea interconexión interna, o red cableada). En cuanto a la funcionalidad, incluye una mayor expresión disponible para los algoritmos concurrentes, proporcionando control no habitual en el paralelismo de datos, o en modelos basados en paralelismo implícito por compilador. En cuanto a prestaciones, especialmente en las CPU modernas, el manejo de la jerarquía de memoria es un punto a tener en cuenta; en el caso del paso de mensajes, deja al programador la capacidad de tener un control explícito sobre la localidad de los datos.

Por contra, el principal problema del paso de mensajes es la responsabilidad que el modelo hace recaer en el programador. Éste debe explícitamente implementar el esquema de distribución de datos, las comunicaciones entre tareas y su sincronización. En estos casos, su responsabilidad es evitar las dependencias de datos, evitar *deadlocks* y condiciones de carrera en las comunicaciones, así como implementar mecanismos de tolerancia a fallos para sus aplicaciones.

Como en el paso de mensajes disponemos de paralelismo explícito, es el programador quien de forma directa controla el flujo de las operaciones y los datos. El medio más usado para implementar este modelo de programación es a través de una librería, que implementa la API de primitivas habitual en entornos de pase de mensajes.

Estas API de paso de mensajes incluyen habitualmente:

- Primitivas de paso de mensajes punto a punto. Desde las típicas operaciones de envío y recepción (*send* y *receive),* hasta variaciones especializadas de éstas.

- Primitivas de sincronización. La más habitual es la primitiva de Barrier, que implementa un bloqueo a todas (o parte) de las tareas de la aplicación paralela.

- Primitivas de comunicaciones colectivas. Donde varias tareas participan en un intercambio de mensajes entre ellas.

- Creación estática (y/o dinámica) de grupos de tareas dentro de la aplicación, para restringir el ámbito de aplicación de algunas primitivas, permitiendo separar conceptualmente unas interacciones de otras dentro de la aplicación.

3.3. Estructuras de programación

En la implementación de aplicaciones distribuidas y paralelas, suelen surgir una clase de estructuras comunes (a veces formalizadas como patrones de programación), que nos identifican clases de algoritmos que solucionan diferen-

tes problemas bajo las mismas estructuras de control. Bajo esta estructura se encapsula una determinada forma de realizar el control y las comunicaciones de la aplicación.

Hay diferentes clasificaciones de estructuras de programación, pero un subconjunto habitual de ellas es:

3.3.1. *Master-worker*

O también denominado *task-farming*, consiste en dos tipos de entidades, el *master* y los múltiples esclavos. El *master* es responsable de descomponer el problema a computar en diferentes trabajos más simples (o en subconjuntos de datos), los cuales distribuye sobre los *workers* y finalmente recolecta los resultados parciales para componer el resultado final de la computación. Los *workers* ejecutan un ciclo muy simple: recibir un mensaje con el trabajo, lo procesan y envían el resultado parcial de vuelta al *master*. Usualmente, sólo hay comunicación entre el *master* y los *workers*. Uno de los parámetros relevantes de este modelo es el número de *workers* utilizados.

Normalmente, este modelo se puede ampliar con una generalización del trabajo a resolver, como una serie de ciclos donde se repite el esquema básico. El *master* dispone por cada ciclo de una cantidad de trabajo por realizar, que envía a los esclavos, éstos reciben, procesan y envían de vuelta al *master*.

Este paradigma en particular posee ciertas características: como son un control centralizado (en el *master*), además suelen darse ciclos bastante semejantes de ejecución, lo que permite prever el comportamiento futuro en posteriores ciclos. Además se pueden tomar medidas eficaces de rendimiento observando los nodos, las cantidades de trabajo enviadas, y los tiempos correspondientes al proceso realizado en los *workers*, determinando cuáles son más eficientes, o por ejemplo, en entornos heterogéneos qué nodos están más o menos cargados de forma dinámica, y así adaptarnos a estas situaciones con diferentes técnicas.

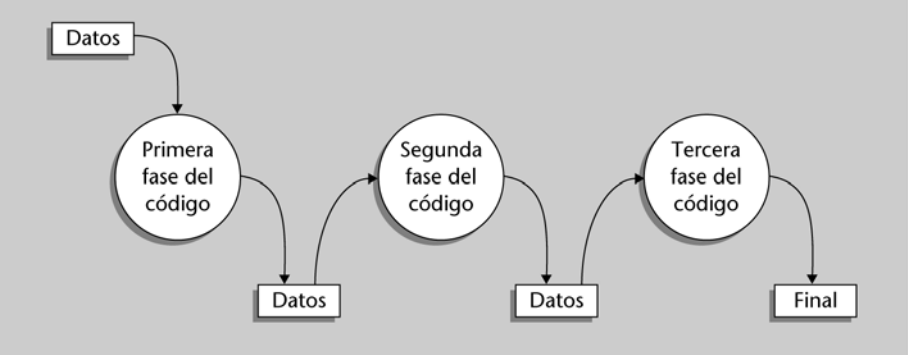
3.3.2. SPMD

En el Single Program Multiple Data (SPMD), se crean un número de tareas idénticas fijas al inicio de la aplicación, y no se permite la creación o destrucción de tareas durante la ejecución. Cada tarea ejecuta el mismo código, pero con diferentes datos. Normalmente, esto significa partir los datos de la aplicación entre los nodos (o máquinas) disponibles en el sistema que participan en la computación. Este tipo de paralelismo también es conocido como paralelismo geométrico, descomposición por dominios o paralelismo de datos.

3.3.3. *Pipelining*

Este paradigma está basado en una aproximación por descomposición funcional, donde cada tarea se tiene que completar antes de comenzar la siguiente, en sucesión. Cada proceso corresponde a una etapa del *pipeline*, y es responsable de una tarea particular. Todos los procesos son ejecutados concurrentemente y el flujo de datos se realiza a través de las diferentes fases del *pipeline*, de manera que la salida de una fase es la entrada de la siguiente. Uno de los parámetros relevantes aquí es el número de fases.

Figura 6



Un cómputo concurrente en *pipeline*

3.3.4. *Divide and conquer*

El problema de computación es dividido en una serie de subproblemas. Cada uno de estos subproblemas se soluciona independientemente y sus resultados son combinados para producir el resultado final. Si el subproblema consiste en instancias más pequeñas del problema original, se puede entonces realizar una descomposición recursiva hasta que no puedan subdividirse más. Normalmente, en este paradigma son necesarias operaciones de partición (*split* o *fork*), computación, y unión (*join*) y trabajamos con dos parámetros, la anchura del árbol –el grado de cada nodo–, y la altura del árbol –el grado de recursión.

3.3.5. Paralelismo especulativo

Si no puede obtenerse paralelismo con alguno de los paradigmas anteriores, se pueden realizar algunos intentos de paralelización de cómputo, escogiendo diferentes algoritmos para resolver el mismo problema. El primero en obtener la solución final es el que se escoge entre las diferentes opciones probadas. Otra posibilidad es: si el problema tiene dependencias de datos complejas, puede intentarse ejecutar de forma optimista, con una ejecución especulativa de la aplicación con posibles vueltas atrás, siempre que aparezcan problemas de incoherencias o dependencias no tenidas en cuenta o previstas que obliguen a deshacer cómputo realizado previamente si algo falla. En caso de que acertemos, o no aparezcan problemas, podemos incrementar de forma significativa el rendimiento.

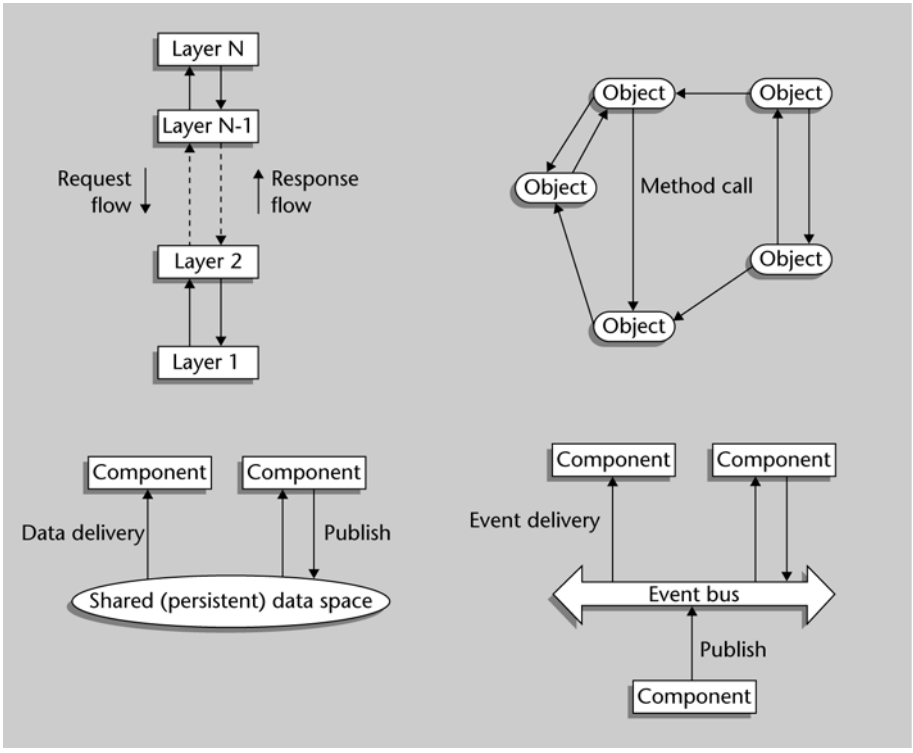
4. Modelos de Interacción

Los diferentes paradigmas de interacción nos proporcionan bases para entender los diferentes modelos que surgen en los sistemas distribuidos.

Como vimos, los estilos arquitectónicos nos describían la organización lógica de los componentes del sistema distribuido. A base de interrelacionar componentes y diferentes tipos de conectores podíamos construir diferentes sistemas distribuidos basados en arquitecturas:

- Por capas
- Orientadas a objetos
- Orientadas a datos
- Orientadas a eventos

Figura 7



Modelos de sistemas distribuidos (capas, objetos, datos, eventos)

También podemos clasificar los sistemas distribuidos en función de la ubicación, jerarquía o relación entre los componentes.

Otro punto de consideración en la creación de sistemas distribuidos es que en muchos casos, la propia infraestructura de base (API o plataforma) sobre la que se desarrolla el paradigma de programación suele ofrecer una serie de servicios asociados, de los que generalmente las aplicaciones suelen utilizar como servicios de base. Constituyendo así un determinado *middleware* para el desarrollo de sistemas distribuidos.

En estos casos suelen encontrarse, entre otros servicios:

- Servicios de nombres: permite enlazar, estableciendo la correspondencia entre algún identificador (como el nombre) con el elemento, componente u objeto requerido.

- Servicios de notificación de eventos: mediante servicios de suscripción, se envían notificaciones de eventos ocurridos en el sistema a los elementos que se hayan suscrito a un tipo particular de suceso, o sobre aquellos que se consideren de interés general.

- Servicios de ciclo de vida: dan facilidades para crear, destruir, copiar o mover elementos en el sistema.

- Servicios de persistencia: permiten recuperar de forma eficiente elementos guardados, seguramente disponiendo de un estado previo.

- Servicios de transacciones: soporta servicios que permiten definir y controlar ciertas operaciones como transacciones (con las propiedades ACID), sincronizando y asegurando la consistencia a los diferentes actores concurrentes.

En las siguientes secciones observamos diferentes modelos de interacción que suelen aparecer en los sistemas distribuidos.

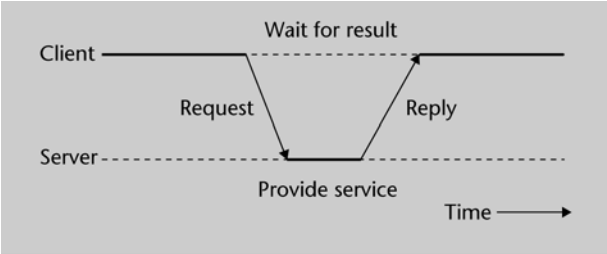
Cabe notar que un determinado sistema puede incluir uno o más de estos modelos, e incluso algunos de ellos están relacionados, ya que en las tareas del sistema, suelen necesitarse uno o más modelos para realizar y desarrollar una determinada funcionalidad. O la misma estructura u organización arquitectural favorece el uso de unos o otros modelos.

4.1. Cliente-servidor

En muchos de los sistemas distribuidos actuales aparecen bajo esta denominación, usando el paradigma de cliente/servidor y los protocolos conocidos como *remote procedure call* (RPC), que son usados normalmente para soportar este paradigma.

La idea básica de esta arquitectura es partir el software en una aplicación para un conjunto de servicios, que proporcionan una serie de operaciones a sus usuarios, y los programas cliente, que implementan aplicaciones y envían las peticiones a los servicios para llevar a cabo las tareas de las aplicaciones.

Figura 8



Modelo básico de interacción cliente-servidor

En este modelo de interacción, los procesos de aplicación no cooperan directamente con otros, pero comparten datos y coordinan acciones interactuando con un conjunto común de servidores, o bien por el orden en que los programas de aplicación son ejecutados.

Encontramos este modelo dentro de sistemas distribuidos en entornos como: sistemas de ficheros, sistemas de bases de datos, servidores de nombres, de tiempo, de seguridad, de correo, de web, y en esquemas más especializados, como los servicios de cómputo en *grid*.

En estos sistemas, los servicios pueden instanciarse en múltiples ocasiones. Normalmente, hablaremos de servicio como un conjunto de servidores que soportan un servicio dado (como los ejemplos anteriores).

Una cuestión particular a tratar en el modelo cliente/servidor son los mecanismos de enlace (*binding*), entre las aplicaciones hacia los servidores que proporcionan el servicio.

Decimos que se produce un enlace cuando un proceso que necesita interaccionar con un servicio distribuido se asocia con un servidor específico que le ofrecerá la respuesta a sus peticiones.

Este enlace suele producirse mediante un protocolo dado entre ambas partes, y mediante intercambios de datos que consolidan el enlace para una o más peticiones. En la mayoría de los servicios, este enlace puede establecerse por mecanismos propios de la definición del servicio, pero también existen mecanismos estándar en algunos entornos.

Por ejemplo, los servicios web (*web services*) tienen un mecanismo estándar de enlace. En este modelo, el sistema cliente comienza buscando un servicio deseado en un espacio de nombres, donde la información sobre los servidores se codifica usando el estándar UDDI, y pueden ser buscados para encontrar una correspondencia exacta, o por un patrón de búsqueda determinado. Una vez encontrado el servició, la aplicación en web services puede realizar una conexión inicial usando TCP, y enviar peticiones de servicio sobre la conexión mediante protocolos como SOAP o codificando peticio-

nes HTTP. En todo caso, esto es posterior al proceso de enlace, y pueden determinarse otras formas de comunicación si las anteriores no resultan eficientes, definiendo nuevos protocolos o escogiendo los más eficientes para una aplicación dada.

Otro punto a tener en cuenta es la replicación de datos, ya que el sistema puede decidir tener más de una única copia de unos datos particulares para permitir accesos locales en múltiples localizaciones, o para incrementar la disponibilidad, en caso de que algunos de los procesos servidores no estén disponibles, bien por fallos o sobrecargas.

Relacionado con lo anterior, también son útiles (o imprescindibles) mecanismos de *caching* para mantener copias locales de datos para acceso rápido si éstos son requeridos más veces. Esto permite beneficiarse de características de los datos, como la localidad de la referencia, y disminuir el ancho de banda en la red para las peticiones o intercambios de datos. Hay que tener en cuenta que este tipo de copia de datos puede estar desactualizada, y por tanto hay que incluir políticas de validación de los datos antes de usarlos. En algunos sistemas, se pueden invalidar datos en *cache* que están siendo actualizados, o refrescarlos de forma explícita. Por el contrario, en otros sistemas, se pueden decidir esquemas más simples, en los cuales los datos son sólo actualizados en ciertos intervalos de tiempo.

El mecanismo principal de la interacción en cliente/servidor son las diferentes formas de RPC usadas, ya sea SOAP en servicios Web, CORBA, o derivados semejantes en las plataformas .NET y J2EE.

Desde el punto de vista del programador, las técnicas RPC son codificadas en un estilo semejante a ciertos tipos de aplicaciones no distribuidas.

De hecho, en RPC, un procedimiento llama a un procedimiento remoto y recibe una respuesta de la misma manera que al realizar una llamada local. En este sentido, RPC aportó a la programación distribuida la abstracción suficiente para conseguir una funcionalidad parecida, sin necesidad de codificar en cada ocasión las comunicaciones necesarias.

Aun así, hay una diferencia fundamental en la programación RPC respecto de las llamadas locales de procedimientos y la separación de la definición de la interfaz del servicio (IDL) o del código que la implementa. En RPC un servicio tiene dos partes. La interfaz define la manera como el servicio será localizado (por nombre, por ejemplo), los tipos de datos que serán usados al lanzar peticiones y las llamadas de procedimientos que soporta. Además de permitir, mediante el IDL, generar el código *stub* para cliente y servidor, siendo éste el encargado en cliente de preparar los mensajes RPC que se en-

viarán por red, y que el *stub* servidor de recibir esta información ejecute la operación solicitada y devolver el resultado.

En general, cuando se define una llamada RPC, intervienen una serie de pasos:

1) El proceso cliente llama a un procedimiento RPC remoto, pasando los parámetros y esperando respuesta.

2) El *stub* cliente crea un mensaje RPC y lo envía al servidor mediante el *runtime* de comunicaciones de la plataforma RPC.

3) El *stub* servidor recibe el mensaje, lo interpreta, llama al procedimiento solicitado con los parámetros.

4) Se ejecuta el procedimiento remoto, obteniéndose el resultado que el *stub* de cliente prepara para devolver al cliente.

5) El *stub* cliente recibe el mensaje, lo interpreta y devuelve al cliente los resultados como si hubiera ejecutado una llamada local.

Dentro de la programación basada en RPC, tenemos una serie de consideraciones extra a tomar en cuenta por parte del programador:

- La mayoría de implementaciones de RPC limitan el tipo de argumentos a utilizar que pueden ser pasados en las llamadas a un servidor remoto, y en algunos casos incluso hay limitaciones de tamaño o de forma en los envíos o en los accesos (por ejemplo, pasar como argumento un puntero a datos no va a significar que estos datos estén disponibles en el espacio de memoria remoto). En la mayoría de los casos se pasan los parámetros por copia, limitando (o eliminando) en algunas implementaciones RPC el uso de punteros. En general, este hecho tiene unas connotaciones importantes en prestaciones, en situaciones donde puede producirse una copia excesiva de datos en las transmisiones.

- El enlace de procedimientos, a diferencia de las llamadas locales, puede fallar en tiempo de ejecución. Procedimientos a los que se llama desde un cliente pueden no obtener enlace cuando el servidor arranca más tarde que el cliente, o sencillamente no está disponible por error. Se necesitará codificar un tratamiento de errores o excepciones en detalle para tener en cuenta los diversos puntos de fallo. Es una tarea que el programador deberá tener en cuenta, y en muchas ocasiones programar basándose en casos de uso.

- La serialización (*marshalling*) de tipos de datos es usada cuando las computadoras en que se ejecutan en cliente y los servidores utilizan distintas representaciones para los datos. El propósito es representar los argumentos

de llamada de manera que sean interpretados por el servidor. En general, el concepto de serialización (*marshalling*) es más amplio, y se refiere a la transformación de la representación en memoria de un objeto o tipo de datos para su almacenamiento o transmisión. Normalmente son usados o bien lenguajes de representación de datos estandarizados, como XML, o API propias creadas para resolver este problema, como por ejemplo las librerías XDR (*eXternal Data Representation*). Los problemas de representación de datos provienen de diversas fuentes, principalmente del soporte del procesador (CPU), donde puede representar de forma diferente: el orden de bytes dentro de un tipo de datos, el alineamiento usado en memoria para colocar el dato, o representaciones concretas para tipos concretos como los números de punto flotante. O sencillamente que uno o más tipos de datos no estén disponibles (o tengan diferentes tamaños) en los dos extremos en comunicación. Generalmente serán necesarios dos procedimientos, en un extremo emisor, un proceso de serialización de los datos (a veces denominado *codificación* o *enpaquetamiento*) a una representación intermedia de uso común, y un proceso, en recepción de deserialización (*unmarshalling*, también llamado *desenpaquetamiento* o *descodificación*), para retornar la representación intermedia a la sistema destino.

- En muchos casos la programación con modelos cliente servidor descansa en la utilización de otros servicios (asociados) proporcionados por la implementación del paradigma de cliente/servidor. Disponiendo de unos servicios básicos que sirven de apoyo, como servicios de nombres, de tiempo, de seguridad, de transacciones...

- En particular, actualmente también se dispone de implementaciones de paquetes de gestión de múltiples hilos de ejecución (*multithreading*), lo que permite combinar el modelo de cliente-servidor con paradigmas de memoria compartida, para dar un mejor soporte concurrente a las múltiples peticiones que puede recibir un servidor, además de poder acceder a los mismos datos compartidos. De hecho, hoy en día, la programación *multithread* es un hecho consumado en la programación de sistemas distribuidos, tanto por razones de aumento de escalabilidad y *troughput* de peticiones atendidas como por el aprovechamiento de los recursos locales del ordenador, en especial en el momento en que comienzan a estar ampliamente disponibles los sistemas *multicore* con soporte en hardware para disponer de hilos realmente concurrentes y no multiprogramados.

Por otra parte, las interacciones entre cliente/servidor siguen diferentes modelos de comunicación, principalmente basados en mecanismos de *acknowledgment* de mensajes.

La idea básica es que el cliente transmitirá su petición y esperará a recibir el ACK correspondiente del servidor, y una vez éste resuelve su petición, le re-

transmitirá los posibles datos de vuelta, y el cliente terminará devolviendo un ACK al servidor.

A este mecanismo básico se le pueden efectuar diferentes optimizaciones, dependiendo de las transmisiones que haya que hacer (dependiendo de éstas y sus tamaños, pueden decidirse a realizar en ráfaga, o empaquetándolas en una única), disminuir el número de ACK, simplemente considerando algunos implícitos, o únicamente los de final de operación. Aunque estas posibles variaciones hay que examinarlas en detalle, dependiendo del entorno utilizado para definir las posibles problemáticas con que pueden encontrarse en entornos adversos, como por ejemplo altas latencias, cortes de canal de transmisión con perdidas de paquetes de datos, etc.

En algunos casos, por tolerancia de fallos, pueden introducirse esquemas multinivel o jerárquicos en el esquema cliente/servidor para adoptar configuraciones de *backup*, donde existan uno o más servidores de *backup* que nos permiten mantener el estatus de un servidor a varios niveles, permitiendo sustituirlo si se produjera fallo o malfuncionamiento temporal. En estos casos entre primarios y servidores de *backup* se suelen duplicar/replicar las operaciones realizadas sobre los servidores primarios.

También hay que añadir en el uso del modelo de interacción cliente/servidor, las connotaciones de *stateless* y *stateful* al modelo de servidor:

- En *stateless*, cada petición recibida es tratada como una transacción independiente, no relacionada con ninguna previa. En este tipo de diseño, hacemos responsable de asegurar el estado y sus acciones a cada una de las partes participantes, cliente y servidores. Por otra parte, simplifica el diseño del servidor porque no es necesario mantener las interacciones pasadas o preocuparse por liberar recursos si hay cambios en los clientes. Normalmente, conlleva incluir más información en las peticiones, y enviar información redundante que necesitará volver a interpretarse en cada petición. Un ejemplo de ello es el servidor web, donde se hacen peticiones (en URL) que especifican e identifican completamente el documento pedido y que no requieren ningún contexto o historia de peticiones previas. Además, la tolerancia de fallos es más sencilla, sencillamente si un servidor falla, mientras sea remplazado (o recuperado) de forma rápida o automática, el efecto es prácticamente no detectable, ya que el nuevo servidor puede responder de forma indistinta a la petición. Por otra parte, examinando la interacción, el cliente debe protegerse cuidadosamente del estado de los datos recibidos, ya que el servidor únicamente promete que los datos son válidos en el momento en que han sido proporcionados. El cliente debe tenerlo en cuenta para determinar si los datos pueden ser no válidos (o caducados) en un momento posterior, en especial cuando se dan condiciones de carrera (*race conditions*). Este tipo de diseño no implica necesariamente que no haya ningún control de estado compartido, de hecho, puede implementarse,

por ejemplo, por mecanismos de *cache* de ciertos datos. Únicamente el funcionamiento no debería necesitar o ser dependiente de esta información compartida o ha de ser especialmente precisa en un momento determinado.

- En el caso *stateful*, la información compartida entre cliente y servidor permitiría que el cliente asumiera acciones locales bajo la consideración de que los datos son correctos. En particular, este diseño requiere en general el uso de mecanismos de bloqueos de información y de protocolos de sincronización. Por ejemplo, si un cliente está modificando datos, se deberían bloquear éstos, potencialmente esperando que finalizasen las lecturas pendientes de otros o escrituras que deban finalizar, modificar entonces los datos, verificarlos si es el caso, y liberar el bloqueo. Normalmente, pueden definirse diferentes políticas y estrategias en estos bloqueos, o colocarlos directamente en exclusiones mutuas en forma más pesimista. Las implementaciones pueden usar mecanismos de bajo nivel ocultos al programador, o bien dejar mecanismos disponibles para éste en forma de control de transacciones. Para tolerancia de fallos, es más problemático el tratamiento de éstos, si hay una caída, se pierde el estado volátil del servidor. Se necesitan mecanismos de recuperación de estado basados en históricos/resúmenes del diálogo con los clientes, o poder abortar operaciones en marcha cuando se produjo la pérdida de servicio. Además, los servidores deben estar informados de los fallos en clientes para poder reclamar los recursos que han aportado, por ejemplo, para mantener el estado de interacción con los clientes perdidos.

Dentro del modelo de interacción cliente-servidor, podemos encontrar diferentes entornos de invocación remota, con mecanismos de codificación binaria de los datos como CORBA, JavaRMI, DCOM, ONC-RPC, y mecanismos donde se usan recientemente codificaciones textuales basadas en XML y el transporte por HTTP, como XML-RPC y SOAP.

4.2. Servicios multiservidor y grupos

En estos modelos trataremos diferentes formas de interacción que surgen en grupos de procesos que tienen una comunicación con diferentes patrones entre ellos.

Un caso especial es el de multiservidores. El servicio es implementado con diferentes servidores que se ejecutan en varias computadoras, y que interactúan para proporcionar un servicio a los procesos clientes, repartiéndose, por ejemplo, las diferentes tareas que componen el servicio, o bien pueden mantener diferentes réplicas de una o más tareas necesarias.

En el caso genérico de grupos, diferentes procesos, relacionados directamente o no, interactúan con varios patrones durante la ejecución en un sistema distribuido, produciéndose diferentes fases de computación y comunicación entre ellos. Dejamos fuera de estos aspectos los sistemas P2P, que, aunque

mantienen esta idea, disponen normalmente de una infraestructura común para una tarea determinada, y se caracterizan por la utilización de recursos aportados por los mismos miembros del grupo.

Normalmente, los grupos se forman sobre la base de proveer de mecanismos para la replicación de datos y la computación concurrente de una o más tareas. Cuando comenzamos a replicar datos, hemos de tener en cuenta que se producirán múltiples accesos concurrentes en más de una replica, con replicas a su vez de cómputo de tareas, o bien con grupos, a su vez diferentes, de tareas en cómputo, pero con posible compartición de datos.

El objetivo final de estas interacciones en grupo es soportar la replicación de datos y la replicación de cómputo para maximizar la alta disponibilidad y mejorar prestaciones.

Si examinamos la posible pertenencia al grupo de uno o más procesos, podemos encontrarnos con grupos:

- Estáticos: el grupo es creado a partir de una lista inicial de miembros, que ya están disponibles en el inicio, o van a estarlo en breves momentos. No se necesitará un control exhaustivo de éstos, pero sí estar atento a posibles errores o desapariciones que podrían provocar el fallo del sistema o de algunas de sus partes (dependiendo del nivel soportado de tolerancia a fallos). En estos sistemas dispondremos, normalmente, de una lista para lectura con todos los procesos que forman el sistema (y posiblemente réplicas de éstos). La lista no cambiará, pero en un momento dado, sólo un subconjunto del grupo puede estar disponible. Aunque pueden también crearse esquemas donde se actualice la lista, por ejemplo *off-line* con nuevos procesos o nodos disponibles. En general, los algoritmos implementados en estos sistemas estáticos se basan en técnicas de quórum para realizar una determinada actividad, lo que nos llevará a prestaciones limitadas, o a baja escalabilidad en número de procesos.

- Dinámicos: en estos grupos suelen introducirse servicios de pertenencia al grupo, en los que se confía para adaptar la lista de pertenencia y controlar las situaciones dinámicas de aparición o desaparición de miembros del grupo. Estos servicios monitorizan el comportamiento dinámico del grupo, pudiendo disponer de información actualizada mucho más rápidamente, y evitando usar quórum o consensos amplios para las operaciones (creándoles importantes *overheads*), permitiendo aumentar en gran manera las prestaciones y por tanto el rendimiento obtenido, aumentando la escalabilidad del sistema fácilmente. En los modelos dinámicos, normalmente se asume que más que el nodo (o computador), se pone el foco en que los procesos son los miembros del sistema, éstos pueden irse o venir, y se modelan los procesos de asociarse al sistema (*join*), o liberarse de éste (*leave*). A veces un sistema por errores o fallos puede liberarse accidentalmente del sistema,

éste debe ser capaz de detectarlo, y, si fuera pertinente, notificarlo a los posibles afectados, o solucionar las problemáticas que puedan aparecer.

Así, en general, los estáticos asumen un conjunto preespecificado de miembros que son capaces de controlar fallos de subconjuntos de forma fácil, pero quizás lenta de cara a prestaciones y escalabilidad. Por otro lado, los dinámicos utilizan servicios de monitorización de pertenencia al grupo para seguir el estado de éste, lo que resulta una forma más flexible de construir mayores arquitecturas, aunque con mayor complejidad algorítmica para solucionar los problemas que aparecen. Es un compromiso que cabe estudiar entre velocidad y flexibilidad por una parte, y complejidad de implementación por otra.

En estos modelos de interacción de grupos, suelen utilizarse diferentes patrones de comunicación entre los participantes, algunos típicos que podemos encontrar son:

- *Multicast*: envíos de datos desde un miembro a un determinado subgrupo de receptores.

- *Broadcast*: a todos los participantes. En muchos sistemas, dependerá su eficiencia de la implementación de los mecanismos. En algunos casos, tanto *multicast* como *broadcast* suelen implementarse como comunicaciones punto a punto, perdiendo los beneficios de implementaciones más eficientes, o el aprovechamiento de recursos hardware dedicados a este tipo de comunicaciones.

- Punto a punto: comunicación desde un participante emisor a otro como receptor.

- *Scatter*: emisión de un conjunto de datos repartiéndose entre los receptores, de forma equitativa o no.

- *Gather*: recolección de un conjunto de datos procedentes de diferentes emisores, apartando cada uno de ellos el mismo tamaño de datos o diferente.

- *Barrier*: bloqueo en ejecución de los participantes hasta que todos, o un subconjunto de ellos, hayan llegado a una zona concreta de código. Este caso en particular puede verse como una forma de sincronización global o local de los participantes.

Por otra parte, en algunos sistemas permiten gestión de participantes, agrupándolos en diferentes grupos separados, por lo que se puede establecer alguna de las interacciones anteriores a nivel de grupo (subconjunto) de los participantes. En diversos sistemas, también esta estructuración puede crearse de forma inicial con grupos estáticos, o de forma dinámica, permitiendo a los participantes entrar, o dejar de formar parte de uno o más grupos. Asimismo, es común, en estos sistemas de grupos, la existencia de un grupo global que

incluye a todos los participantes y, por otra parte, los diferentes grupos estáticos o dinámicos que pueden crearse por separado.

En la mayoría de entornos (API o *midlewares*) de procesamiento por grupo (o para multiservidor) se introducen una serie de interfaces típicas:

- Balanceo de carga, que permitan manejar más trabajo a medida que el número de miembros del grupo crezca.
- Ejecución garantizada, de manera que sea transparente al usuario (si un miembro no está disponible, se intenta recuperarlo o bien sustituirlo por uno equivalente).
- Bloqueos, se proporcionan sincronización o alguna interfaz para paso de *tokens*.
- Datos replicados y la forma de leerlos y escribirlos, así como garantizar diferentes grados de integridad en los datos.
- *Logs*, se mantienen resúmenes de la actividad y puntos de restauración (*checkpoints*), para proporcionar reproducción y posibles vueltas atrás.
- Acceso a nodos remotos, maneras de integrar nodos aislados o redes locales en una solución más amplia (WAN o Internet).
- *Ranking* de miembros, para examinar posibilidades de subdividir tareas, o realizar trabajos de balanceo de carga.
- Monitorización y control, para instrumentar las comunicaciones en un grupo, y para controlar aspectos de las comunicaciones, así como analizar el comportamiento o las prestaciones del sistema.
- Transferencia de estado, permitir transferir el estado actual a un nuevo miembro que se una al grupo.

El modelo de interacción de multiservidor y/o grupos de procesos es muy utilizado en sistemas de Internet para permitir, entre otras, aplicaciones de:

- Réplica de servidores: aquellas que necesitan ofrecer disponibilidad, tolerancia a fallos o balanceos de carga.
- Diseminación de datos: se necesitan transmisiones de datos a grupos de participantes, con actualizaciones de éstos de forma dinámica o con modelo *publish-subscribe*.
- Gestión del sistema: propagar información monitorizada de sistema, detección de fallos y coordinación de acciones de recuperación.

- Aplicaciones de seguridad: actualización dinámica de políticas de seguridad, replicación de datos de seguridad para disponibilidad.

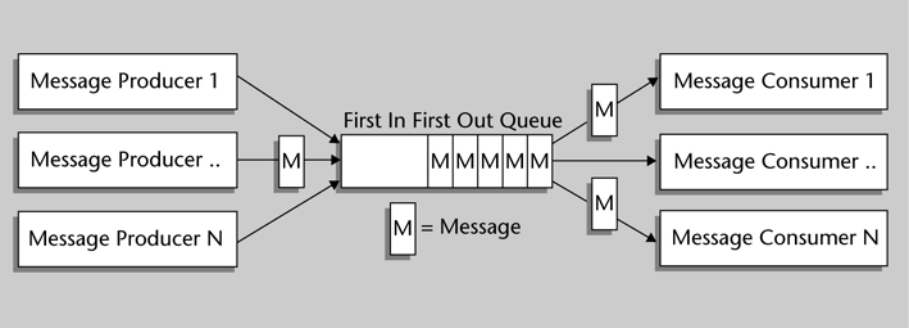
Podemos citar algunos ejemplos, como los servicios de directorio Sun NIS (*Network Information Services*), usando datos replicados para controlar el acceso de *login* a un conjunto de computadores. También los *clusters* de alta disponibilidad para web para soportar por ejemplo buscadores o tiendas en línea.

4.3. Arquitecturas basadas en mensajes

En este modelo, generalmente incluyendo principalmente los MOM (*message oriented middleware*), se basan en software de comunicación entre aplicaciones, que generalmente se apoya en el paradigma de paso de mensajes asíncrono, en oposición a la metáfora de emisor/receptor.

Normalmente, dependen de la existencia de sistemas de colas de mensajes, aunque algunas implementaciones también se basan en sistemas de *multicast* y *broadcast*.

Figura 9



Estructura básica de cola en MOM

Se suelen proporcionar también opciones de seguridad en el acceso (sólo las aplicaciones permitidas pueden actuar de receptor o emisor en las colas), gestión de prioridad (mensajes con diferentes niveles de urgencia), control de flujo (para controlar tamaño de las colas bajo situaciones extremas), balanceo de la carga (si diversos procesos se nutren de las mismas colas), tolerancia a fallos (por si las aplicaciones son de larga duración), y persistencia de datos.

Las ventajas principales de este modelo se basan en las comunicaciones orientadas a mensajes, y en su habilidad de guardar, direccionar y transformar los mensajes durante el proceso de entrega de los mismos.

Normalmente estos sistemas proporcionan almacenamiento para salvaguardar el medio de transmisión de mensajes, de manera que los emisores y receptores no

tienen por qué estar presentes a la vez para que la entrega sea realizada. En particular, es interesante este modelo de interacción en situaciones donde se deben manejar conexiones intermitentes, como en:

- Redes no fiables
- Usuarios casuales
- Conexiones temporales

En este sentido, si las aplicaciones receptoras fallan por cualquier razón, las emisoras pueden continuar sin verse afectadas, los mensajes simplemente se acumularan en el almacenamiento, para procesarse posteriormente, cuando los receptores vuelvan a estar disponibles.

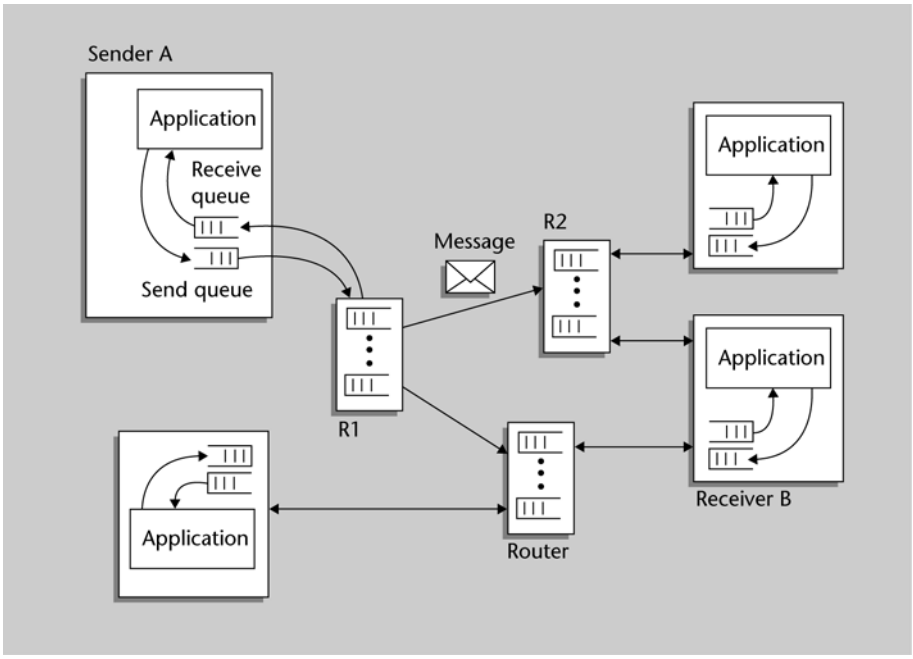
También en el caso de direccionamiento, pueden utilizarse diferentes estrategias para la entrega, ya sean de tipo *multicast* o *broadcast*, o la búsqueda de los caminos correctos o más eficientes para la llegada al receptor.

En el caso de la transformación de mensajes, el *middleware* de MOM ofrece cierta inteligencia para la transformación de formatos, de manera que los extremos de la comunicación dispongan de los mensajes en sus formatos nativos. En algunas implementaciones MOM se ofrecen herramientas visuales que permiten realizar estas transformaciones en forma de interfaz de usuario con operaciones simples de usuario.

En cuanto a algunas desventajas de estos sistemas, podemos citar: no utilización de estándares, normalmente de tipo propietario. La utilización de componentes extra, como el agente de entrega de mensajes, que puede incurrir en penalizaciones de prestaciones (de hecho, en algunos seudo-MOM llega a utilizarse el correo electrónico directamente como agente, o como metáfora del servicio). Por otra parte, un problema principal se da con los modelos de interacción de petición-respuesta (por ejemplo, parecidos a los de cliente servidor), que sean inherentemente síncronos, esperando el emisor la respuesta, ya que el modelo en MOM es asíncrono por naturaleza. Algunos sistemas MOM integran la posibilidad de unir, agrupándolas, la petición y la respuesta en una transacción seudo-síncrona.

Podemos citar algunas implementaciones del modelo MOM, como JMS (*Java message service*) que en particular intenta abstraerse de un MOM concreto mediante el ofrecimiento de una API lo suficientemente genérica (utilizada en Java EE). Entre otros productos propietarios, podemos citar: Microsoft MSMQ, IBM MQseries, HP MessageQ, o los servicios de notificación de eventos de CORBA. En particular estas implementaciones suelen utilizarse en comunicaciones entre aplicaciones cliente de entorno usuario a aplicaciones ejecutándose en *mainframes*.

Figura 10



Organización de MOM con sistemas de colas y enrutamiento entre ellas.

Otros ejemplos que citaremos más adelante hacen referencia al paradigma de paso de mensajes, especialmente utilizado en la programación paralela, mediante API con primitivas de envío y recepción, en diferentes modos de comunicación, ya sea punto a punto o colectivas entre los participantes en la aplicación (esquemas idénticos a los observados anteriormente en grupos). Algunas API dedicadas al paso de mensajes, como PVM e MPI, también hacen uso de colas de mensajes locales a los procesos para el envío y recepción de información (no existiendo colas globales para el sistema).

Por otra parte, los modelos de interacción basados en orientación a servicios disponen comúnmente de subsistemas de mensajería cuando no usan directamente algún MOM.

4.4. Servidores *proxy*

En este modelo, el *proxy* actúa como proceso intermedio en las operaciones que son pasadas desde peticiones de clientes a una cierta configuración de servidores (ya sea cliente-servidor puro o múltiples servidores).

Normalmente suele utilizarse en las diferentes arquitecturas, dependiendo de la escalabilidad necesaria para atender a un número elevado de clientes, o por mantener altos índices de disponibilidad de servicio.

Pueden darse diferentes características dependiendo del servicio:

- Un servicio de *proxy* podría verse como un conjunto (con variadas arquitecturas) de *proxies* que disponen de *caches* intermedias de los datos.

- En casos *stateless* (como la web), el servidor no dispone de mecanismos para avisar a la configuración de los *proxies* de que los datos primarios han cambiado. En lugar de esto, la configuración de *proxies* debe refrescar periódicamente los datos que manejan, volviendo a preguntar por los mismos datos por si han cambiado.

- En todo caso, es especialmente crítica la coherencia de la información, ya que los datos pueden estar replicados en múltiples nodos del *proxy*.

- Hay que tener en cuenta que en casos como la web, se establecen una serie de compromisos, se intenta maximizar la disponibilidad a costa de disponer de información menos actualizada o incluso incoherente. En el caso de la web, es una problemática más importante a medida que los contenidos se hacen más dinámicos, lo que implicará modificar el modelo de *proxy stateless* por uno más dinámico, que maximice la coherencia de los datos.

- Los *proxie*s permiten reducir la carga de servicio respecto de los servidores, e incluso pueden llegar a sustituirlos en la respuesta en caso de fallos, inaccesibilidad o sobrecarga del servidor.

Un ejemplo típico de *proxy* es el *web proxy server*, que proveen de *caches* compartidas de recursos web para diferentes máquinas clientes. También algunos *proxy* actúan de intermediarios a *web servers* remotos, como control adicional de seguridad y/o *firewall*.

4.5. Código móvil

En este modelo consideramos entornos muy dinámicos, donde posiblemente el conjunto de usuarios, los dispositivos y los elementos software se esperan que cambien de forma frecuente.

Siendo alguna de las características habituales de este modelo (y de hecho problemas a tratar):

- Cómo aparecen los componentes para asociarse, y cómo operan con otros elementos cuando se mueven, fallan o espontáneamente aparecen.

- Cómo se integran los sistemas con el mundo físico en un determinado contexto, y cómo se les permite interactuar con sensores y actuadores.

- La seguridad y privacidad a tratar en sistemas físicos volátiles.

- Técnicas para adaptarse a dispositivos de tamaño reducido con limitaciones de recursos de cómputo y E/S.

La computación móvil y ubicua ha surgido principalmente a partir de los factores de miniaturización y las conexiones sin cables (*wireless*).

Aunque también el paradigma móvil es usado ampliamente en situaciones distribuidas con alto dinamismo de participantes, y con recursos limitados de algún tipo, o bien para adaptarse por parte de los participantes a un entorno especialmente variable, para aportar tolerancia a fallos, balanceo de carga, o simplemente transmisión de datos y cómputo al entorno adecuado.

Los problemas habituales con que estos modelos han de desenvolverse:

- Asociación: el código (y/o su soporte físico o lógico, ya sea dispositivo o plataforma software de computación) pueden aparecer o desaparecer de forma impredecible. Por ejemplo, en el caso de una red, el dispositivo (u el código) deberán adquirir acceso a la red, y obtener direcciones y registrar su presencia. Serán necesarios también servicios de descubrimiento para recibir qué servicios o recursos están disponibles, y posiblemente conocer qué interfaz es necesaria para acceder a ellos. Por ejemplo, un posible modelo de API puede hacer referencia a un servicio o un determinado recurso aportado o pedido:

  – Servicio = registrar (*dirección*, atributos). Registrar un servicio en una determinada dirección con los parámetros que lo definan.

  – Refrescar (*servicio*). Refrescar el servicio por ejemplo delante de cambios, o redefinición de este.

  – DesRegistrar(*servicio*). Liberar el servicio.

  – ConjuntoServicios = Preguntar (*Requisitos*). Retornar un conjunto de servicios que correspondan con los requisitos establecidos.

- Interoperabilidad: capacidad en un sistema volátil de que uno o más elementos se asocien y se interconecten, bien para finalmente producir una interacción de tipo cliente servidor, o de uso/ofrecimiento de recursos, o en la cooperación para el desarrollo de una computación. Normalmente basándose en API estándar, o plataformas comunes (o en su alternativa mediante posibles pasarelas), para establecer la asociación, compartición de recursos, y comunicación.

Este paradigma móvil puede verse especialmente en determinados entornos, como la web con los *applets* como código móvil que se desplaza desde un servidor al cliente para ejecutarse en su plataforma (ya sea ordenador) y/o dispositivo móvil, adaptándose así a las diferentes plataformas y dispositivos. O como entornos pensados intrínsecamente basándose en los principios de asociación, petición (o aporte) y localización (de servicios o recursos), e interoperabilidad como es el caso de la plataforma Jini.

También los conceptos de virtualización permiten ofrecer nuevas posibilidades, mediante código móvil para la tolerancia a fallos de servicios, la alta disponibilidad de éstos, o el balanceo óptimo de peticiones.

O como el caso de Java (y .NET) para la realización de máquinas virtuales sencillas, que permiten adaptarse fácilmente a diferentes plataformas de computadores y sistemas operativos o dispositivos móviles, permitiendo una transportabilidad del código en sistemas heterogéneos. La aproximación por máquina virtual provee de mecanismos para permitir la ejecución en cualquier hardware, ya que los compiladores de un lenguaje particular generan código para la máquina virtual en lugar de código particular para un hardware concreto.

4.6. Procesamiento *peer-to-peer*

En los modelos de P2P, hay dos características que los han separado de orígenes compartidos con modelos cliente-servidor, y el de interacción de grupos.

Por una parte, la fusión en el proceso o nodo del sistema de características por una parte de cliente, y por otras de servidor, a la vez integrados en el mismo participante, de manera que por un lado puede verse como cliente de unos servicios determinados y soportar (simultáneamente) servicios de cara otros participantes.

Por otro lado, la cualidad de soportar únicamente su funcionamiento conjunto por los recursos aportados por los mismos participantes del grupo, sin depender (en principio) de recursos externos al mismo.

A pesar de esto último, en general el modelo es más completo, para abarcar diferentes propuestas que parten con hasta tres clases de participantes diferenciados:

- Los clientes auténticos que normalmente son externos al sistema, conectándose a otros miembros del sistema de alguna manera pero que no participan activamente de los protocolos asociados a P2P.

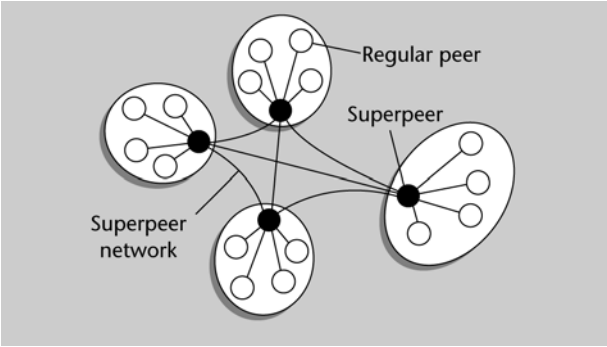
- Los *peers* son clientes que, por diversas razones, es habitual que estén en el sistema, tengan una disponibilidad temporal, de recursos y de conexión suficiente para participar activamente en los protocolos.

- Finalmente, los servidores son participantes pertenecientes por la entidad que provee del servicio P2P son clientes muy estables que han sido promovidos a roles más importantes dentro del servicio P2P. En algunos casos se les denomina *superpeers*, y varios clientes *peer* están conectados a ellos, creando estructuras jerárquicas.

Dependiendo de la arquitectura (recordemos las diferencias entre los P2P estructurados o no), no todos los sistemas dispondrán de las tres categorías de

nodos/participantes. Pero en algunos sistemas sí que pueden existir y participar en ellos con diferentes roles y protocolos.

Figura 11



Una organización P2P jerárquica con *superpeers*

En general, la aproximación de alto nivel es considerar (en especial en los estructurados) que todos los nodos son iguales, permitiendo comportamiento simultáneo de cliente y servidor por parte de cada participante. Con este comportamiento simétrico, en el modelo P2P se cuestiona cómo organizar los procesos en la red *overlay*, esto es, la red que está formada por los nodos como procesos y los enlaces representando posibles canales de comunicaciones. La red *overlay* puede crearse tanto de forma estructurada como no.

En el caso estructurado, la red se crea mediante el uso de DHT (*distributed hash tables*) para organizar los procesos. En un sistema basado en DHT, a los ítems de datos se les asigna una clave aleatoria (desde un espacio de identificadores amplio, por ejemplo 128 o 160 bits). Asimismo, se identifica de forma semejante a los nodos del sistema desde el mismo espacio. El tema crucial en cada implementación de DHT es cómo se acaba mapeando de forma única la clave de un dato a un identificador de nodo según métricas de distancia. O cómo se obtiene, buscando un dato, la dirección de red del nodo responsable de éste.

Típicamente, en el caso DHT aparecen operaciones sobre el *middleware* P2P del tipo:

- Put (GUID, data): los datos se guardan en réplica en todos los nodos responsables para un objeto identificado por GUID.

- Remove (GUID): borra todas las referencias a GUID y los datos asociados.

- Value=get (GUID): los datos asociados con GUID son recuperados desde uno de los nodos responsables de ellos.

Por el contrario, en el caso no estructurado, la construcción de la red de *Overlay* se basa en algoritmos aleatorios. La idea principal es disponer en cada nodo de una lista de sus vecinos, pero esta lista es construida de forma aleatoria, ya que se asume que los datos están colocados aleatoriamente en nodos; de modo que, cuando un nodo necesita localizar un dato específico, lo que

hay que hacer es desplegar en la red una petición de búsqueda. En muchos casos lanzando peticiones en *broadcast* o en *multicast* de manera jerárquica explorando el espacio de nodos. En concreto, estas formas costosas de búsqueda pueden ser optimizadas dependiendo de la existencia de réplicas, facilitando así diferentes caminos para encontrar los datos buscados. En algunos casos, estas réplicas suceden de forma natural por el propio funcionamiento de la red, como por ejemplo en el caso de compartición de ficheros.

Con los casos no estructurados, se suele mantener un tipo de API denominado DOLR (*distributed object location and routing*), con esta interfaz, los objetos pueden almacenarse en cualquier parte y la capa DOLR es responsable de mantener el *mapping* entre los identificadores de objeto y las direcciones de los nodos donde se mantienen réplicas de los objetos. Permitiendo operaciones en el *middleware* como:

- *Publish* (GUID): se computa el identificador del objeto. Publicando en el *host* el objeto correspondiente al identificador.

- *Unpublish* (GUID): dispone que el objeto sea inaccesible.

- *SendToObj* (msg, GUID,[n]): envío de un mensaje o llamada al GUID correspondiente, o si aparece el parámetro *n*, a las *n* replicas de éste.

Los modelos de P2P han sido utilizados para implementar una gran cantidad de funcionalidades como: a los ya mencionados servicios de compartición de ficheros como las redes de Napster (en el histórico modelo anterior), Gnutella, Kazaa o CAN. Servicios de indexado distribuido como Chord, Pastry implementando mecanismos de DHT (*distributed hash table*). Computación distribuida en base a recursos intermitentes, como el caso de SETI@home, y la plataforma de desarrollo más genérica BOINC. Otros usos más recientes utilizan los P2P como sistemas de distribución de contenidos para vídeo y radio en línea (por ejemplo, IPTV), o como soporte de sistemas de vídeo bajo demanda (VoD).

Nota

El proyecto JXTA es una tecnología de código abierto (Open Source) para el desarrollo de aplicaciones P2P.

https://jxta.dev.java.net

Por otra parte, para la programación de sistemas P2P cabe destacar el *framework* Java JXTA para el desarrollo de sistemas P2P. JXTA es una plataforma P2P y proporciona *middleware* no ligado a tareas específicas (como compartición de ficheros). En lugar de esto se proporciona un *middleware* general usable para una variedad de aplicaciones. Como detalles particulares, permite la creación de grupos de *peers*, formando subredes de *peers* de la global, permitiendo solapamiento o inclusión de unos grupos en otros; por otro lado, existen *superpeers* de tipo *rendezvous* para coordinación de *peers* y propagación de mensajes (en especial si están en diferentes redes), y los *relay peer* que se utilizan cuando algunos *peers* están detrás de sistemas NAT o *firewall*, para permitir comunicar mensajes a través de ellos.

4.7. Arquitecturas orientadas a servicios

Las arquitecturas orientadas a servicios son una tendencia para la interconexión de sistemas heterogéneos. Se ofrecen básicamente una serie de servicios de aplicación débilmente acoplados y altamente interoperables, basando la comunicación entre ellos y las aplicaciones subyacentes, mediante el uso de una plataforma de interacción independiente, y lenguajes u entornos basados en conceptos de servicios. Asimismo, los objetivos principales son la reutilización de los elementos de aplicación usados, así como que los servicios puedan ser usados en más de un contexto en diferentes momentos.

En algunos modelos de las arquitecturas orientadas a servicios, uno de los componentes principales es el bus de mensajes, donde las aplicaciones pueden subscribirse a diferentes tópicos y los protocolos basados en este bus tienden a optimizarse para altas velocidades, usando hardware para el *broadcast* cuando sea posible. Los mensajes intentan entregarse tan pronto como sea posible al proceso de aplicación.

Además, las arquitecturas orientadas a servicios suelen basarse principalmente en interacciones por mensajes o invocaciones a llamadas, mediante codificación textual en XML, con la pretensión de solucionar los problemas de interoperabilidad que se plantean en sistemas heterogéneos debido al uso de diferentes plataformas de servicios basadas en esquemas de cliente/servidor. En la mayor parte de estos casos, se tenían que utilizar sistemas complejos de puentes y pasarelas para conectarlos. De esta manera, los sistemas orientados a servicios suelen basarse en protocolos abiertos e interoperables como HTTP para el transporte de datos y XML para la codificación de los datos.

En particular han tenido una gran aceptación los sistemas basados en los servicios web basándose en estándares de comunicación como SOAP, que permite interoperabilidad entre sistemas basándose en HTTP y codificación XML.

Así, en estas arquitecturas se acaba definiendo un servicio como un conjunto de mensajes de entrada enviados a un objeto o a una composición de objetos, junto con el retorno producido en forma de mensajes de salida.

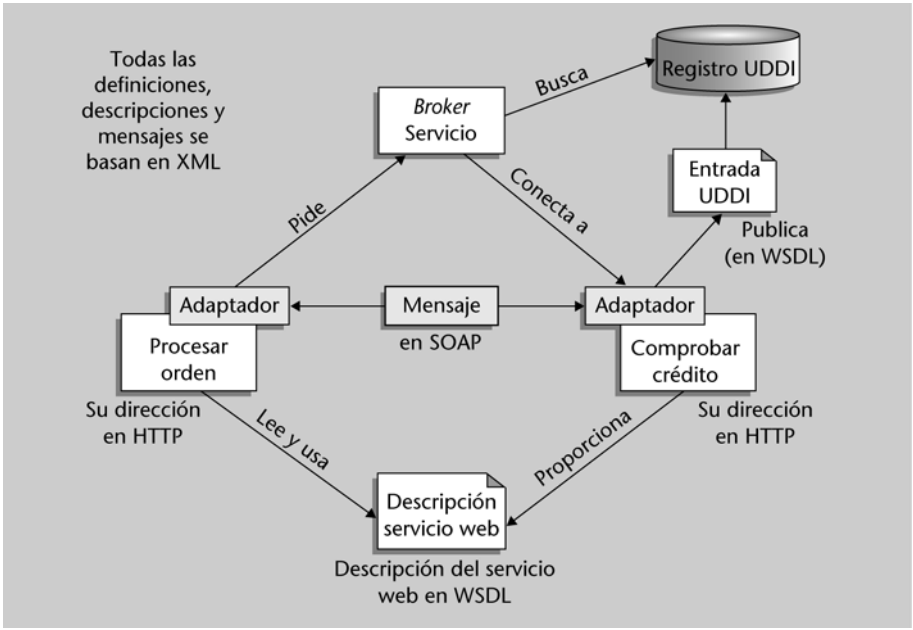
En general, hoy día se usa el termino SOA (*service oriented architecture*) para definir una combinación de tecnologías, en las que se usa XML para codificar los datos de forma independiente a la plataforma, los *web services*, como método de integración de sistemas heterogéneos (en especial SOAP para intercambiar información), y una estructura de comunicación basado en MOM para proporcionar comunicaciones asíncronas, poco acopladas y flexibles. También se suele utilizar el concepto de bus de comunicación (ESB, *enterprise service bus*),

Nota

Las organizaciones SOAInstitute.org y SOAprinciples.com ofrecen artículos e información sobre sistemas SOA y sus características.

como elemento de conexión de los múltiples componentes, de manera que permitan comunicarse.

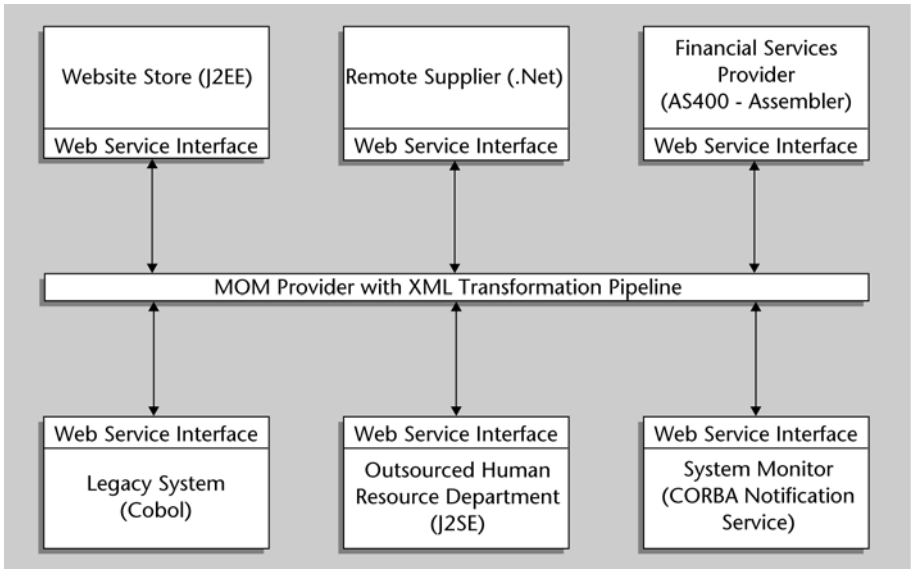
Figura 12



Ejemplo de diversas tecnologías involucradas en SOA

Habilitamos así con esta combinación de tecnologías la posibilidad de reducir el procesamiento en las aplicaciones a cajas negras, que utilizan las tecnologías anteriores para comunicarse e intercambiar información y acceder a servicios. Diseñando de este modo sistemas abiertos con posibilidades de extensión y reutilización.

Figura 13



Un sistema SOA que interconecta subsistemas en múltiples plataformas

Los grandes fabricantes adaptan las arquitecturas SOA con vistas a interconectar sus sistemas heredados (tipo *mainframes*) con clientes y entornos distribuidos en una o varias empresas diferentes (ofreciendo diferentes servicios en

heterogéneas plataformas), para proporcionar soporte para e-comercio en variantes como B2B (entre empresas) o B2C (entre empresa y clientes). Diferentes fabricantes, como IBM, HP, BEA, Oracle, Microsoft o SAP ofrecen soluciones variadas para SOA, y algunas soluciones de código abierto, como OpenSOA (con Jboss).

5. Paradigmas de programación

El término *paradigma* se origina en la palabra griega παράδειμα (*paradeigma*), que significa 'modelo' o 'ejemplo'. En el ámbito de la informática y más específicamente en la generación de código fuente, un paradigma de programación representa un enfoque particular o las cuestiones de índole filosófica para la construcción del programa (*software*). Saber cuándo se debe evaluar qué paradigma hay que utilizar significa muchas veces el principal escollo a superar del proyecto, ya que, una vez escogido cambios posteriores en el mismo, significarán retrasos en tiempo y posibles pérdidas económicas en el desarrollo o reducción de prestaciones en el código desarrollado. NO se debe tener en mente que un paradigma es mejor que otro, sino que cada uno tiene sus ventajas y también sus desventajas, por lo cual la decisión es muy importante y existen un conjunto de situaciones donde un paradigma es más adecuado que otro. Algunos ejemplos comunes de paradigmas son:

- Paradigma imperativo o procedural: es el más común y es el que se utiliza en lenguajes como C o Pascal.

- Paradigma funcional: donde se representa un conjunto de reglas declarativas basándose en reglas matemáticas. Uno de los lenguajes que lo soportan es Lisp.

- Paradigma lógico: se basa en la utilización de un corpus de conocimiento para la aplicación de reglas lógicas. Prolog es uno de los lenguajes que permite la utilización de este paradigma.

- Paradigma orientado a objetos: este paradigma utiliza objetos software que permiten la encapsulación de datos y propiedades dentro del mismo. Un lenguaje que soporta este tipo de paradigmas es C++ o Java.

Muchas veces, y dependiendo de los objetivos del proyecto, puede seleccionarse una forma pura de paradigma para programar las aplicaciones, pero también es habitual que se mezclen dando lugar a programación multiparadigma.

En el caso de sistemas paralelos y/o distribuidos, existen un conjunto de paradigmas adaptados para explotar la potencialidad de hardware subyacente. Teniendo en cuenta las definiciones enumeradas en los capítulos anteriores, podemos tener en cuenta diferentes consideraciones sobre los paradigma/modelo de programación:

1) Esfuerzo: existen diferencias importantes del esfuerzo invertido en escribir programas paralelos en relación con su eficiencia y dependencia, en algunos casos, sobre la arquitectura subyacente.

2) Diferentes grados y forma de paralelismo de las aplicaciones que se adecuan más o menos a los diferentes lenguajes de programación.

3) Tipo de paralelismo: en el algoritmo paralelo se debe especificar explícitamente cómo trabajan los procesadores para implementar la solución específica. En este caso, la tarea del compilador es simple, pero la del programador/depurador de la aplicación es compleja. Implícito es aquel paralelismo que resuelve el compilador de la aplicación que normalmente está escrita en forma secuencial para generar, así, código paralelo, que se ejecutará en una máquina multiprocesador.

4) Arquitectura de memoria: el espacio de memoria puede ser compartido o disjunto, por lo cual será necesario adecuar el paradigma de programación a un espacio de memoria común, donde los programadores ven la aplicación como un conjunto de procesos accediendo a una zona de memoria compartida con los restantes. En caso de memoria distribuida, cada proceso ve sólo su parte de la memoria e interactúa con los otros procesos enviando y recibiendo mensajes (paso de mensajes) para intercambiar información. En el primer caso, se pueden dar problemas de coherencia de datos y en el segundo, dependiendo de cómo sea el envío o recepción, se pueden perder datos o bloquearse los procesos si la espera se alarga. Normalmente, los sistemas del primer tipo se pueden programar también como del segundo.

Entre los paradigmas más conocidos para la programación de computadores paralelos/distribuidos, podemos mencionar: paso de mensajes, memoria compartida, tareas, paralelismo de datos, operaciones remotas en memoria, objetos distribuidos, MOM (*message oriented middleware*), *web services* o modelos combinados. Es importante destacar que no son excluyentes entre sí y en la mayoría de los casos no dependen de la arquitectura subyacente.

Es importante no confundir los paradigmas de programación paralela con los esquemas o modelos de programación de soluciones paralelas (algoritmos o esquemas genéricos) y que, generalmente, dependen de diferentes aspectos vinculados a los datos, como por ejemplo paralelismo, particionado, estructuración, función objetivo, etc.

5.1. Paso de mensajes

El paso de mensajes es una técnica empleada habitualmente en programación de tareas (procesos) concurrentes, permitiendo que éstos se puedan sincronizar (ejecutar o esperar en un orden determinado) y permitir la exclusión mutua entre los mismos (zonas de acceso a recursos compartidos excluyentes entre todos los procesos que deseen acceder en forma dinámica). Su principal característica es que no necesita memoria compartida entre los procesos que

tienen que intercambiar información, por lo cual, es uno de los paradigmas más comunes en la programación de sistemas distribuidos.

Si consideramos el proceso como el elemento básico de cómputo (es decir, el código binario de la aplicación, datos estáticos y dinámicos, pila y estructura de datos del sistema operativo –PCB–), es éste el que incluirá las sentencias de envío y recepción de los mensajes para comunicarse con otros procesos (normalmente distribuidos en otros procesadores de la arquitectura).

Dependiendo de si el proceso que envía el mensaje espera a que el mensaje sea recibido, se puede hablar de paso de mensajes bloqueantes (síncronos) o no bloqueantes (asíncronos). En el paso de mensajes no bloqueantes, el proceso que envía no espera a que el mensaje sea recibido, y el proceso que recibe ejecuta la llamada, y si no ha llegado el mensaje continúa su ejecución (pudiéndose dar el caso de que el mensaje llegue pero que nunca sea leído por el receptor). En el caso de envío no bloqueante, generalmente se combina con la utilización de buzones (*mailbox*) en el receptor para evitar así la pérdida de mensajes y que el código de quien lo envía vaya a la máxima velocidad posible (hasta que se sature el buzón del receptor). En el paso de mensajes síncrono (bloqueante), el proceso que envía el mensaje espera a que un proceso lo reciba para continuar su ejecución y el que recibe se espera hasta que haya un mensaje para leer. Esta técnica es conocida como técnica encuentro (*rendezvous*) y sus procesos se ejecutarán a la velocidad del proceso más lento.

Normalmente, este paradigma se implementa a través de dos primitivas (*send* y *receive*) que tienen la siguiente estructura:

- *send*(*message, messagesize, target, type, flag*)

– *message* contiene los datos a enviar,
– *messagesize* indica el tamaño en bytes,
– *target* es el identificador del procesador o procesadores destino,
– *type* es una constante con la que se distingue entre varios tipos de mensajes, y
– *flag* indica si la operación es bloqueante o no bloqueante.

- *receive*(*message, messagesize, source, type, flag*)

– *messsage* indica el lugar donde se guardarán los datos,
– *messagesize* indica el número máximo de bytes a recibir,
– *source* indica la etiqueta del procesador del cual se recibirá el mensaje,
– *type* especifica el tipo de mensaje que se va a leer, ya que puede existir más de un mensaje en el buzón de comunicación y el parámetro type selecciona un determinado mensaje para leer,
– *flag* especifica si la operación de recibir es bloqueante o no bloqueante.

La interfaz de paso de mensajes (conocida como MPI, *message passing interface*) es un protocolo de comunicación entre ordenadores y es el estándar para la co-

municación entre los procesadores que ejecutan una aplicación en un sistema de memoria distribuida. La definición de la interfaz de programación (API) de MPI ha sido el resultado del trabajo del MPI *Fourum* (MPIF), que es un consorcio de más de cuarenta organizaciones. MPI tiene influencias de diferentes arquitecturas, lenguajes y trabajos en el mundo del paralelismo, como son: WRC (Ibm), Intel NX/2, Express, nCUBE, Vertex, p4, Parmac y contribuciones de ZipCode, Chimp, PVM, Chamaleon, PICL. El principal objetivo de MPIF fue diseñar una API, sin relación particular con ningún compilador ni biblioteca tal que permitiera la comunicación eficiente (*memory-to-memory copy*), cómputo y comunicación concurrente y descarga de comunicación –siempre y cuando exista un coprocesador de comunicaciones– y además, que soportara el desarrollo en ambientes heterogéneos, con interfaz C y F77 (incluyendo C++, F90), donde la comunicación fuera fiable y los fallos fueran resueltos por el sistema. La API también debía tener interfaz para diferentes entornos (PVM, NX, Express, p4...), disponer una implementación adaptable a diferentes plataformas con cambios insignificantes y que no interfiera con el sistema operativo (*thread-safety*). Esta API fue diseñada especialmente para programadores que utilizaran el *message passing paradigm* (MPP) en C y F77 para aprovechar la característica más relevante: la portabilidad. El MPP se puede ejecutar sobre máquinas multiprocesadores, redes de WS e incluso sobre máquinas de memoria compartida. La versión MPI1 (la versión más extendida) no soporta creación (*spawn*) dinámica de tareas, pero MPI2 (en creciente evolución) sí que lo hace.

Muchos aspectos han sido diseñados para aprovechar las ventajas del hardware de comunicaciones sobre SPC (*scalable parallel computers*) y el estándar ha sido aceptado mayoritariamente por los fabricantes de hardware paralelo y distribuido (SGI, SUN, Cray, HPConvex, IBM, Parsystec...). Existen versiones *freeware* (por ejemplo, MPICH) que son totalmente compatibles con las implementaciones comerciales realizadas por los fabricantes de hardware e incluyen comunicaciones punto a punto, operaciones colectivas y grupos de procesos, contexto de comunicaciones y topología y un entorno de control, administración y *profiling*. Pero existen también algunos puntos no resueltos, como son: operaciones de memoria compartida (SHM), ejecución remota, herramientas de construcción de programas, depuración, control de *threads*, administración de tareas, funciones de entrada/salida concurrentes (la mayor parte de estos problemas de falta de herramientas están resueltos en la versión 2 de la API MPI2). El funcionamiento en MPI1, al no tener creación dinámica de procesos, es muy simple, ya que hay tantos procesos como tareas existan, autónomos y que ejecutan su propio código estilo MIMD (*multiple instruction multiple data*) y comunicándose vía llamadas MPI. El código puede ser secuencial o *multithread* (concurrentes) y MPI funciona en modo *threadsafe*, es decir, se pueden utilizar llamadas a MPI en *threads* concurrentes, ya que las llamadas son reentrantes.

Un aspecto interesante de MPI es que soporta comunicaciones colectivas, por ejemplo:

MPI_Barrier( ): bloquea los procesos hasta que la ejecutan todos.

MPI_Bcast( ) *broadcast* del proceso raíz a todos los demás.
MPI_Gather( ) recibe valores de un grupo de procesos.
MPI_Scatter( ) distribuye un *buffer* en partes a un grupo de procesos.
MPI_Alltoall( ) envía datos de todos los procesos a todos.
MPI_Reduce( ) combina valores de todos los procesos.
MPI_Reduce_scatter( ) combina valores de todos los procesos y distribuye.
MPI_Scan( ) reducción prefija (0,...,i-1 a i).

5.1.1. Ejemplos de programación

Para compilar programas MPI, se puede utilizar el comando *mpicc* (por ejemplo, *mpicc* –o *test test.c*), que acepta todas las opciones de *gcc*, aunque es recomendable utilizar (con modificaciones) algunos de los *makefiles* que se hallan en los ejemplos (por ejemplo, en Linux Debian en /usr/doc/mpich/examples). También se puede utilizar *mpireconfig Makefile,* que utiliza como entrada el archivo *Makefile.in* para generar el *makefile* y es mucho más fácil de modificar. Después se podrá hacer:

mpirun -np 8 *programa*

o bien:

mpirun.mpich -np 8 *programa*

donde *np* es el número de procesos o procesadores en los que se ejecutará el programa (8, en este caso). Se puede poner el número que se desee, ya que Mpich intentará distribuir los procesos en forma equilibrada entre todas las máquinas de **/etc/mpich/machines.LINUX**. Si hay más procesos que procesadores, Mpich utilizará las características de intercambio de tareas de GNU/Linux para simular la ejecución paralela. En Debian Linux y en el directorio /usr/doc/mpich-doc (un enlace a /usr/share/doc/mpich-doc) se encuentra toda la documentación en diferentes formatos (comandos, API de MPI, etc.).

- Para compilar MPI: mpicc -O –o output output.c

- Ejecutar Mpich: mpirun.mpich -np Nº_procesos output

El comando mpirun permite ejecutar un programa bajo Mpich, pero existe otra implementación de MPI llamada LAM MPI para diferentes arquitecturas (por ejemplo, Linux) con prestaciones similares. Si LAM MPI está instalado, utiliza el comando mpirun para realizar la misma acción y es por ello que mpich utiliza mpirun.mpich para diferenciarse del anterior.

A continuación veremos dos ejemplos (que se incluyen con la distribución Mpich 1.2.x en el directorio /usr/doc/mpich/examples). *Srtest* es un programa

simple para establecer comunicaciones entre procesos punto a punto, y *cpi* calcula el valor de Pi forma distribuida (por integración).

Comunicaciones punto a punto: srtest.c

Para compilar: **mpicc -O –o srtest srtest.c**

Ejecución Mpich: **mpirun.mpich -np *N.º_procesos* srtest** (para evitar que se solicitela contraseña [*N.º procesos* - 1] veces si no se tiene el acceso directo por *ssh*).

Ejecución LAM: **mpirun -np *N.º_procesos* srtest** (debe ser un usuario diferente de *root*)

```
#include "mpi.h"
#include <stdio.h>
#define BUFLEN 512
int main(int argc, char *argv[]) {
      int myid, numprocs, next, namelen;
      char buffer[BUFLEN], processor_name[MPI_MAX_PROCESSOR_NAME];
      MPI_Status status;
      MPI_Init(&argc,&argv);
            /* Debe ponerse antes de otras llamadas MPI, siempre */
      MPI_Comm_size(MPI_COMM_WORLD,&numprocs);MPI_Comm_rank(MPI_COMM_WORLD,&myid);
            /*Integra el proceso en un grupo de comunicaciones*/
            MPI_Get_processor_name(processor_name,&namelen);
            /*Obtiene el nombre del procesador*/
      fprintf(stderr,"Proceso %d sobre %s\n", myid, processor_name); strcpy(buffer,"Hola Pueblo");
      if (myid ==numprocs1) next = 0;
      else next = myid+1;
      if (myid ==0) { /*Si es el inicial, envía string de buffer*/.
               printf("%d Envío '%s' \n",myid,buffer);
            MPI_Send(buffer, strlen(buffer)+1, MPI_CHAR, next, 99, MPI_COMM_WORLD);
            /*Blocking Send, 1°:buffer, 2°:size, 3°:tipo, 4°:destino, 5°:tag, 6°:contexto*/
            /*MPI_Send(buffer, strlen(buffer)+1, MPI_CHAR,
            MPI_PROC_NULL, 299,MPI_COMM_WORLD);*/
            printf("%d recibiendo \n",myid);
            /* Blocking Recv, 1°:buffer, 2°:size, 3°:tipo, 4°:fuente, 5°:tag, 6°:contexto, 7°:status*/
            MPI_Recv(buffer, BUFLEN, MPI_CHAR, MPI_ANY_SOURCE, 99, MPI_COMM_WORLD,&status);
            printf("%d recibió '%s'\n",myid,buffer) }
      else {
            printf("%d recibiendo \n",myid);
            MPI_Recv(buffer, BUFLEN, MPI_CHAR, MPI_ANY_SOURCE, 99,
            MPI_COMM_WORLD,status);
            /*MPI_Recv(buffer, BUFLEN, MPI_CHAR, MPI_PROC_NULL,
            299,MPI_COMM_WORLD,&status);*/
            printf("%d recibió '%s' \n",myid,buffer);
            MPI_Send(buffer, strlen(buffer)+1, MPI_CHAR, next, 99,
            MPI_COMM_WORLD);
            printf("%d envió '%s' \n",myid,buffer);}
      MPI_Barrier(MPI_COMM_WORLD); /*Sincroniza todos los procesos*/
      MPI_Finalize(); /*Libera los recursos y termina*/ return (0);
      }
```

Cálculo de PI distribuido: *cpi.c*

Para compilar: ***mpicc -O* -o *cpi cpi.c*.**

Ejecución Mpich: **mpirun.mpich -np *N.º procesos* cpi** (para evitar que se solicite la contraseña de las máquinas donde se ejecutará cada uno de los procesos es conveniente tener conexión por *ssh* sin contraseña y con llave pública/privada).

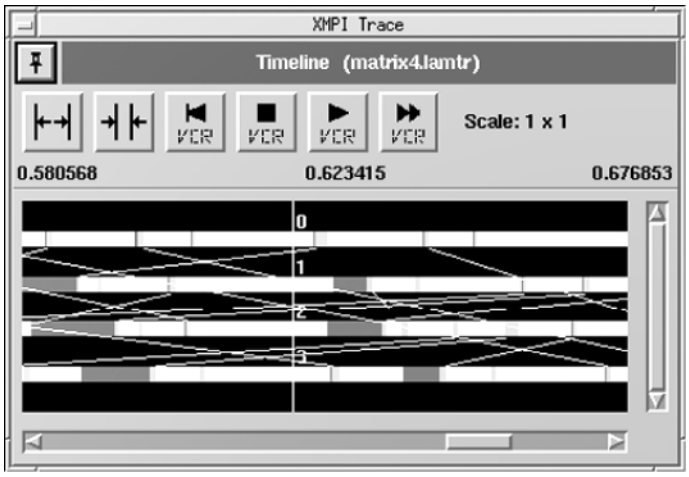
Ejecución LAM: **mpirun -np *N.º procesos* cpi** (debe ser un usuario diferente de *root*).

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
double f( double );
double f( double a) { return (4.0 / (1.0 + a*a)); }

int main( int argc, char *argv[] ) {
            int done = 0, n, myid, numprocs, i;
      double PI25DT = 3.14159265358979323846462643;
      double mypi, pi, h, sum, x;
      double startwtime = 0.0, endwtime;
      int namelen;
      char processor_name[MPI_MAX_PROCESSOR_NAME];

      MPI_Init(&argc,&argv);
      MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
            /*Indica el número de procesos en el grupo*/
      MPI_Comm_rank(MPI_COMM_WORLD,&myid); /*Id del proceso*/
      MPI_Get_processor_name(processor_name,&namelen);
            /*Nombre del proceso*/
      fprintf(stderr,"Proceso %d sobre %s\n", myid, processor_name);
      n = 0;
      while (!done) {
            if (myid ==0) { /*Si es el primero...*/
                  if (n ==0) n = 100; else n = 0;
                  startwtime = MPI_Wtime();} /* Time Clock */
      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
            /*Broadcast al resto*/ /*Envía desde el 4.° arg. a todos
      los procesos del grupo. Los restantes que no son 0
      copiarán el buffer desde 4 o arg -proceso 0-*/
      /*1.°:buffer, 2.° :size, 3.° :tipo, 5.° :grupo */
      if (n == 0) done = 1; else {
            h = 1.0 / (double) n;
            sum = 0.0;
            for (i = myid + 1; i <= n; i + = numprocs) {
                  x = h * ((double)i - 0.5); sum + = f(x); }
            mypi = h * sum;
      MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
      /* Combina los elementos del Send Buffer de cada proceso del
      grupo usando la operación MPI_SUM y retorna el resultado en
      el Recv Buffer. Debe ser llamada por todos los procesos del
      grupo usando los mismos argumentos*/ /*1.° :sendbuffer, 2.°
      :recvbuffer, 3.° :size, 4.° :tipo, 5.° :oper, 6.° :root, 7.°:contexto*/
            if (myid == 0){ /*Sólo el P0 imprime el resultado*/
            printf("Pi es aproximadamente %.16f, el error es %.16f\n", pi, fabs(pi -
            PI25DT));
            endwtime = MPI_Wtime();
            printf("Tiempo de ejecución = %f\n", endwtime-startwtime); }
            }
      }
      MPI_Finalize(); /*Libera recursos y termina*/
      return 0;
}
```

Un aspecto interesante para el programador, además de la depuración que puede realizar a través de herramientas como el gdb y su interfaz gráfica xxgdb o el ddd, es que puede visualizar la ejecución de programas paralelos para obtener información sobre la dinámica del mismo. En MPI existe una herramienta llamada XMPI (en Debian *xmpi*), que permite visualizar la ejecución de una aplicación distribuida. También es posible instalar una biblioteca, *libxmpi3*, que implementa el protocolo XMPI para analizar gráficamente programas MPI

con más detalles que los ofrecidos por *xmpi*. La figura siguiente muestra unas de las posibles gráficas de xmpi.

Figura 14



5.1.2. *Message-oriented middleware* (MOM)

MOM comprende una categoría de comunicación entre aplicaciones de software que, en general, se basa el paso de mensajes en forma asíncrona, frente a una filosofía de petición/respuesta. La mayoría de *middleware* orientado a mensajes depende de una cola de mensajes del sistema, aunque algunas implementaciones dependen de sistemas de emisión *(broadcast*) o de sistemas de mensajería de multidifusión (*multicast*).

MOM puede considerarse como una extensión natural del paradigma de paso de mensajes en la capa más baja de la red modelo OSI. A diferencia de RPC y de orientación a objetos, es una forma de comunicación asincrónica, es decir, el remitente no bloquea la espera a que los receptores participen en el intercambio de datos. Si la persistencia del servicio de mensajes ofrece fiabilidad, el receptor no es necesario que esté en funcionamiento cuando el mensaje es enviado. A diferencia de OOM (*object oiented mddleware*), en MOM, los mensajes son en general sin tipo y la estructura interna de los mensajes es responsabilidad de la aplicación.

En MOM tradicionales, los mensajes son dirigidos a sus destinatarios, aunque el emisor y el receptor están desacoplados y no es necesario sincronizar la comunicación. Esto puede ser poco adecuado para sistemas de gran escala (*wide-area*, *large-scale*) pero puede ser ventajoso para desacoplar las fuentes y receptores de mensajes con respecto a la identificación (*naming*) y así se pueden establecer comunicaciones anónimas o sin conocer al interlocutor. Una típica forma de describir esta comunicación es la llamada *sistemas de publicar-suscribir*, donde las fuentes "publican" en toda la red, y los consumidores interesados se "suscriben" a los mensajes. Los mensajes sólo

serán enviados a la red si hay al menos un suscriptor solicitante. Esto requiere que el servicio de transporte de mensajes pueda comprender aspectos internos del mismo, aunque algunos sistemas son "*topic-based*", donde cada mensaje tiene una línea de "asunto" que puede leer el sistema de transporte e ignorar el resto del mensaje.

MOM tiene una gran cuota de mercado (mayor que *object oriented middleware*), y se utiliza para acceso a bases de datos en grandes aplicaciones empresariales. Entre los proyectos más interesantes de MOM, se pueden enumerar:

- IBM Corporation: WebSphere MQ
- Sun: Sun Java Message Service (JMS), Sun^{TM} ONE Middleware
- Microsoft: MSMQ Microsoft Message Queue Server
- ObjectWeb (Open Source): JORAM
- TIBCO Software Inc: TIBCO Rendezvous.
- Message Queuing for C++ (open Source) MQ4CPP

5.2. RPC

Las RPC (*remote procedure call*, llamada a procedimiento remoto) es un protocolo que permite a un programa de ordenador ejecutar código en otro ordenador remoto sin tener que preocuparse por las comunicaciones entre ambos. El protocolo es un gran avance sobre los BSD *sockets* usados hasta el momento y de esta manera el programador no tiene que estar pendiente de las comunicaciones, estando éstas encapsuladas dentro de las RPC. Las RPC son muy utilizadas dentro del paradigma de programación cliente-servidor, siendo el cliente el que inicia el proceso, solicitando al servidor que ejecute cierto procedimiento o función y enviando este último el resultado de dicha operación al cliente. Hay distintos tipos de RPC, muchos de ellos estandarizados, como pueden ser las RPC de Sun denominado ONC RPC (RFC1057), las RPC de la OSF llamadas DCE/RPC y el modelo de objetos de componentes distribuidos de Microsoft DCOM. Un problema grave que existe entre todas estas implementaciones es que ninguna de ellas es compatible con las otras y la mayoría de ellas utilizan un lenguaje de descripción de interfaz (IDL) que define los métodos exportados por el servidor. En la actualidad se utiliza XML como lenguaje para definir el IDL y HTTP como protocolo de red, dando lugar a lo que se conoce como *servicios web*.

Las RPC permiten integrar la filosofía cliente-servidor con la programación procedural, habilitando a los clientes a comunicarse con los servidores en forma similar al uso de las llamadas SO convencionales, y las llamadas a los procedimientos remotos son modeladas como una llamada convencional, pero el procedimiento se ejecuta en otro proceso y generalmente otra máquina.

5.2.1. Ejemplos de programación

A continuación se verán unos ejemplos de programación en Open Network Computing RPC (Sun), que es la implementación más habitual de RPC. El paquete RPC de Sun incluye un compilador llamado *rpcgen*, que genera los esqueletos y las interfaces automáticamente, utiliza XDR (eXternal Data Representation) para hacer el *marshalling/unmarshalling* de los datos enviados entre cliente y servidor, incluye tres tipos de datos básicos (int, float, char) y permite un lenguaje declarativo para especificar tipos de datos complejos.

Fecha y hora remota: date.x

Ejecutando *rpcgen date.x* generará date.h, date_clnt.c y date_svc.c. El archivo de cabecera (date.h) debe ser incluido tanto en el cliente como con el servidor.

```
/* date.x Especificación de la fecha y hora remota */
/* Definimos dos procedimientos:
            bin_date_1() retorna la fecha y hora en binario
            str_date_1() recibe hora binaria y retorna un string */
program DATE_PROG {
     version DATE_VERS {
            long BIN_DATE(void) = 1; /* Procedimiento número = 1 */
            string STR_DATE(long) = 2; /* Procedimiento número = 2 */
     } = 1;     /* Versión número = 1 */
} = 0x2345678; /* Programa número = 0x2345678 */
```

Siempre iniciar la numeración desde 1 (procedimiento 0 es siempre un "proceso nulo"). El número de programa es un número definido por el usuario y se aconseja utilizar el rango entre 0x20000000 a 0x3fffffff, así como también se debe poner siempre el prototipo para cada función. Sun RPC permite sólo un único parámetro y un único resultado, y si fueran necesarios más parámetros/resultados, se debe utilizar una estructura (ampliar información sobre utilización de XDR). En este caso se utilizará la función clnt_create() para obtener un descriptor (*handle*) al procedimiento remoto.

```
/* rdate.c programa cliente para el programa de fecha remoto */
#include <stdio.h>
#include <rpc/rpc.h> /* standard RPC include */
#include "date.h" /* este archivo es generado por rpcgen */


main(int argc, char *argv[]) {
  CLIENT *cl; /* RPC handle */
  char *server;
  long *lresult; /* valor de retorno de bin_date_1() */
  char **sresult; /* valor de retorno de str_date_1() */

  if (argc != 2) {
  fprintf(stderr, "usage: %s hostname\n", argv[0]);
  exit(1); }

  server = argv[1]; /* Servidor pasado como primer argumento */
  /* Creo el descriptor del cliente (client handle) */
    if ((cl = clnt_create(server, DATE_PROG, DATE_VERS, "udp")) == NULL) {
    clnt_pcreateerror(server); /* No puedo establecer conexión con el servidor. Fin */
   exit(2); }

  /* Primero llamo al procedimiento "bin_date" */
```

```
    if ( (lresult = bin_date_1(NULL, cl)) == NULL) {
    clnt_perror(cl, server); /* Error. Fin */
    exit(3); }
  printf("Fecha-Hora sobre el host %s = %ld\n",server, *lresult);

   /* Ahora llamo al procedimiento str_date */
     if ( (sresult = str_date_1(lresult, cl)) == NULL) {
      clnt_perror(cl, server); /* Error. Fin*/
      exit(4); }
   printf("Fecha-Hora sobre el host %s = %s", server, *sresult);
   clnt_destroy(cl); /* Libero el descriptor */
   exit(0); /* Fin */
}

/* dateproc.c Procedimiento remoto llamado por el server stub */
#include <rpc/rpc.h> /* standard RPC include */
#include "date.h" /* archivo generado por rpcgen */

/* Retorna la fecha y hora en binario */
long *bin_date_1() {
  static long timeval; /* debe ser static */
  timeval = time((long *) 0);
  return(&timeval);
}
/* Convierte hora binaria en un formato string */
char **str_date_1(long *bintime) {
   static char *ptr; /* debe ser static */
   ptr = ctime(bintime); /* convierte a local time */
   return(&ptr);
}
```

Como se puede observar en estas rutinas, no dispone de una función main(), ya que serán llamadas por el código *date_svc.c* generado por *rpcgen*. Las declaraciones de variables de retorno deben ser estáticas para que el valor sea correcto (si no ellas serán alojadas en el *stack* cuando la función retorne). Los tres ficheros necesarios serán date.x dateproc.c rdate.c. Para compilar y generar los *stubs*:

- rpcgen date.x

Esto generará los siguientes ficheros (además de los que teníamos): date.h date_clnt.c date_svc.c. Para compilar debo ejecutar:

- cc -o rdate rdate.c date_clnt.c
- cc -o dateproc dateproc.c date_svc.c

Para ejecutar el servidor (en *background*) hacemos:

- dateproc &

Y luego ejecutamos el cliente donde *server* es el nombre del servidor; por ejemplo, si la máquina donde he ejecutado el servidor se llama *nteum*, hago:

- *rdate nteum*

El resultado será:
Fecha-Hora sobre el *host* nteum = 998776958
Fecha-Hora sobre el *host* nteum= Mon Nov 5 16:40:00 2007

La secuencia de acciones son: el *server* crea un *socket* UDP, lo asocia un puerto local y llama a la función svc_register() para registrar el número de programa y la versión en el proceso *Port mapper* (portmap). Este proceso deberá estar en marcha antes de ejecutar la aplicación (normalmente se pone en marcha al arranque del sistema operativo). A partir de este momento, el servidor espera por peticiones. El cliente contacta cuando se ejecuta con el *portmap* sobre la máquina indicada utilizando UDP, recibe el puerto del servidor y llama a la función prototipo de la función remota (primero bin_date_1() y después str_date_1()), recibe los valores desde el servidor y los imprime.

5.3. Memoria compartida. Modelos de hilos (*threading*).

Normalmente, en una arquitectura cliente-servidor, los clientes solicitan a los servidores determinados servicios y esperan que éstos le contesten con la mayor eficacia posible. Para sistema distribuidos con servidores con una carga muy alta (por ejemplo, sistemas de archivos de red, bases de datos centralizadas o distribuidas), el diseño del servidor se convierte en una cuestión crítica para determinar el rendimiento general del sistema distribuido. Un aspecto crucial en este sentido es encontrar la manera óptima de manejar la E/S, teniendo en cuenta el tipo de servicio que ofrece, el tiempo de respuesta esperado y la carga de clientes. No existe un diseño predeterminado para cada servicio y escoger el correcto dependerá de los objetivos/restricciones del servicio y las necesidades de los clientes. Las preguntas que debemos contestar antes de elegir un determinado diseño son: ¿cuánto tiempo se tarda en un proceso de solicitud del cliente? ¿Cuántas de esas solicitudes es probable que lleguen durante ese tiempo? ¿Cuánto tiempo puede esperar el cliente? ¿Cuánto afecta esta carga del servidor a las prestaciones del sistema distribuido?

5.3.1. MultiThreading

Las últimas tecnologías en programación para este tipo aplicaciones (y así lo demuestra la experiencia) es que los diseños más adecuados son aquellos que utilizan modelos de multihilo (*multithreading models*), donde el servidor en este caso tiene una organización interna de de procesos paralelos o hilos cooperantes y concurrentes.

Un *thread* significa una secuencia de ejecución (hilo de ejecución) de un programa, es decir, diferentes partes o rutinas de un programa informático que se ejecutan concurrentemente en un único procesador. ¿Qué ventajas aporta esto respecto a un programa secuencial? Consideremos que un programa tiene tres rutinas A, B, C. En un programa secuencial la rutina C no se ejecutará hasta que se hayan ejecutado A y B. Si en cambio A, B, C son hilos (*threads*), las tres rutinas se ejecutarán concurrentemente, y si en ellas hay E/S, tendremos

concurrencia de ejecución con E/S del mismo programa (proceso), lo cual mejorará notablemente las prestaciones de dicho programa. Generalmente, los *threads* están contenidos dentro de un proceso y diferentes hilos de un mismo proceso pueden compartir algunos recursos, mientras que diferentes procesos, no. La ejecución de múltiples hilos en paralelo necesita el soporte del sistema operativo y en los procesadores modernos existen optimizaciones del procesador para soportar el *multithreading*.

Generalmente, existen cuatro modelos de diseño por *threads* (en orden de complejidad creciente):

- Un *thread* y un cliente: en este caso el servidor entra en un bucle sin fin escuchando por un puerto y ante la petición de un cliente se ejecutan los servicios en el mismo *thread*. Otros clientes deberán esperar a que termine el primero. Es fácil de implementar, pero sólo atiende a un cliente por vez.

- Un *thread* y varios clientes con selección: en este caso el servidor utiliza un solo *thread* pero puede aceptar múltiples clientes y multiplexar el tiempo de CPU entre ellos. Se necesita una gestión más compleja de los puntos de comunicación (*sockets*), pero permite crear servicios más eficientes, aunque presenta problemas cuando los servicios necesitan alta carga de CPU.

- Un *thread* por cliente: es probablemente el más popular. El servidor espera por peticiones y crea un *thread* de servicio para atender a cada nueva petición de los clientes. Esto genera simplicidad en el servicio y alta disponibilidad, pero el sistema no escala con el número de clientes y puede saturar el sistema muy rápidamente, ya que el tiempo de CPU dedicado ante una gran carga de cliente se reduce notablemente y la gestión del SO puede ser muy compleja.

- Servidor con *threads* en granja (*workers threads*): este método es más complejo, pero mejora la escalabilidad de los anteriores. Existe un número fijo de *threads* (*workers*) a los cuales el *thread* principal distribuye el trabajo de los clientes. El problema de este método es la elección del número de *threads* (*workers*): con muchos, caerán las prestaciones del sistema por saturación, con pocos, el servicio será deficiente (los clientes deberán esperar). Normalmente, será necesario sintonizar la aplicación para trabajar con un determinado entorno distribuido.

Existen diferentes formas de expresar a nivel de programación con *threads*: paralelismo a nivel de tareas o paralelismo a través de los datos. Elegir el modelo adecuado optimiza el tiempo necesario para modificar, depurar y sintonizar el código. La solución a esta disyuntiva es describir la aplicación en términos de dos modelos basados un trabajo en concreto:

– Tareas paralelas cuando *threads* independientes pueden atender a tareas independientes de la aplicación. Estas tareas independientes serán encapsuladas en *threads* que se ejecutarán asincrónicamente y se deberán utilizar librerías como Win32 Thread API (Windows) o POSIX Threads (Unix), las cuales han sido diseñadas para soportar concurrencia a nivel de tarea.

– Modelo de datos paralelos para calcular lazo intensivos, es decir, la misma operación debe repetirse un número elevado de veces (por ejemplo, comparar una palabra con las palabras de un diccionario). Para este caso, es posible encargar la tarea al compilador de la aplicación, o si no es posible que el programador describa el paralelismo utilizando el entorno OpenMP, que es una API que permite escribir aplicaciones eficientes bajo este tipo de modelos.

Una aplicación de información personal (*personal information manager*) es un buen ejemplo de un aplicación que contiene concurrencia a nivel de tareas (p. ej. acceso a la base de datos, libreta de direcciones, calendario...). Esto podría ser en seudo código:

```
    Function addressBook;
    Function inBox;
    Function calendar;
Program PIM {
    CreateThread (addressBook);
    CreateThread (inBox);
    CreateThread (calendar);}
```

Un buen ejemplo de operaciones con paralelismo de datos es un corrector de ortografía, por ejemplo en seudocódigo:

```
Function SpellCheck {
    loop (word = 1, words_in_file) compareToDictionary (word); }
```

Se debe tener en cuenta que ambos modelos (*threads* paralelos y datos paralelos) pueden existir en la misma aplicación. A continuación se mostrará el código de un productor de datos y un consumidor de datos basado en *pthread*. Para compilar sobre Linux, por ejemplo, hay que utilizar: gcc –o pc pc.c –lptread.

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define QUEUESIZE 10
#define LOOP 20

void *producer (void *args);
void *consumer (void *args);
typedef struct { /* Estructura del buffer compartido y descriptores de
threads */
    int buf[QUEUESIZE]; long head, tail; int full, empty;
    pthread_mutex_t *mut; pthread_cond_t *notFull, *notEmpty;
queue;
queue *queueInit (void); /* Prototipo de función: inicialización del
buffer */
void queueDelete (queue *q); /* Prototipo de función: borrado del
buffer*/
```

```
void queueAdd (queue *q, int in); /* Prototipo de función: insertar
elemento en el buffer */
void queueDel (queue *q, int *out); /* Prototipo de función: quitar
elemento del buffer */

int main () {
    queue *fifo; pthread_t pro, con; fifo = queueInit ();
    if (fifo == NULL) { fprintf (stderr, "Error al crear buffer.\n");
exit (1); }
    pthread_create (&pro, NULL, producer, fifo); /* Creación del thread
    productor */
    pthread_create (&con, NULL, consumer, fifo); /* Creación del thread
    consumidor*/
    pthread_join (pro, NULL); /* Espera del main() hasta que terminen
    ambos threads */
    pthread_join (con, NULL);
    queueDelete (fifo); /* Eliminación del buffer compartido */
    return 0; } /* Fin */

void *producer (void *q) { /*Función del productor */
    queue *fifo; int i;
    fifo = (queue *)q;
    for (i = 0; i < LOOP; i++) { /* Insertamos en el buffer
    elementos=LOOP*/
    pthread_mutex_lock (fifo->mut); /* Semáforo para entrar a insertar
    */
    while (fifo->full) {
        printf ("Productor: queue FULL.\n");
        pthread_cond_wait (fifo->notFull, fifo->mut); }
    /* Bloqueamos del productor si el buffer está lleno, liberando el
semáforo mut para que pueda entrar el consumidor. Continuará cuando el
consumidor ejecute pthread_cond_signal (fifo->notFull);*/
    queueAdd (fifo, i); /* Inserto elemento en el buffer */
    pthread_mutex_unlock (fifo->mut); /* Libero el semáforo */
    pthread_cond_signal (fifo->notEmpty);/*Desbloqueamos consumidor si
    está bloqueado*/
    usleep (100000); /* Dormimos 100 mseg para permitir que el consumidor
    se active */
    }
    return (NULL); }

void *consumer (void *q) { /*Función del consumidor */
    queue *fifo; int i, d;
    fifo = (queue *)q;
    for (i = 0; i < LOOP; i++) { /* Quito del buffer elementos=LOOP*/
        pthread_mutex_lock (fifo->mut); /* Semáforo para entrar a quitar
        */
        while (fifo->empty) {
            printf ("Consumidor: queue EMPTY.\n");
            pthread_cond_wait (fifo->notEmpty, fifo->mut); }
/* Bloqueamos el consumidor si el buffer está vacío, liberando el
semáforo mut para que pueda entrar el productor. Continuará cuando el
consumidor ejecute pthread_cond_signal (fifo->notEmpty);*/
            queueDel (fifo, &d); /* Quitamos elemento del buffer */
        pthread_mutex_unlock (fifo->mut); /* Liberamos el semáforo */
        pthread_cond_signal (fifo->notFull); /*Desbloqueamos el
        productor si está bloqueado*/
            printf ("Consumidor: Recibido %d.\n", d);
            usleep(200000);/* Dormimos 200 mseg para permitir que el
            productor se active */
   }
   return (NULL); }

queue *queueInit (void) {
    queue *q;
    q = (queue *)malloc (sizeof (queue)); /* Creación del buffer */
    if (q == NULL) return (NULL);
    q->empty = 1; q->full = 0; q->head = 0; q->tail = 0;
    q->mut = (pthread_mutex_t *) malloc (sizeof (pthread_mutex_t));
    pthread_mutex_init (q->mut, NULL); /* Creación del semáforo */
    q->notFull = (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
```

```
    pthread_cond_init (q->notFull, NULL); /* Creación de la variable
    condicional notFull*/
    q->notEmpty = (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
    pthread_cond_init (q->notEmpty, NULL); /* Creación de la variable
    condicional notEmpty*/
    return (q); }+

void queueDelete (queue *q) {
    pthread_mutex_destroy (q->mut); free (q->mut);
    pthread_cond_destroy (q->notFull); free (q->notFull);
    pthread_cond_destroy (q->notEmpty); free (q->notEmpty);
    free (q); }

void queueAdd (queue *q, int in) {
    q->buf[q->tail] = in; q->tail++;
    if (q->tail == QUEUESIZE) q->tail = 0;
    if (q->tail == q->head) q->full = 1;
    q->empty = 0;
    return; }

void queueDel (queue *q, int *out){
    *out = q->buf[q->head]; q->head++;
    if (q->head == QUEUESIZE) q->head = 0;
    if (q->head == q->tail) q->empty = 1;
    q->full = 0;
    return; }
```

5.3.2. OpenMP

El OpenMP (Open - Multi Processing) es una interfaz de programación de aplicaciones (API) con soporte multiplataforma para la programación en C/C++ y Fortran de procesos utilizando memoria compartida sobre plataformas Unix y Windows. Esta infraestructura se compone de un conjunto de directivas del compilador, rutinas de la biblioteca y variables de entorno que permiten aprovechar recursos compartidos en memoria y en tiempo de ejecución.

Definidos conjuntamente por un grupo de los principales fabricantes de hardware y software, OpenMP permite utilizar un modelo escalable y portátil de programación proporcionando a los usuarios una interfaz simple y flexible para el desarrollo, sobre plataformas paralelas, de aplicaciones de escritorio a aplicaciones de altas prestaciones sobre *supercomputers*. Una aplicación construida con el modelo híbrido de la programación paralela puede ejecutarse en un ordenador utilizando tanto OpenMP como *Message Passing Interface* (MPI).

OpenMP es una implementación *multithreading*, mediante la cual un *thread maestro* divide la tareas sobre un conjunto de *threads trabajadores*. Estos *threads* se ejecutan simultáneamente y el entorno de ejecución realiza la asignación de éstos a los diferentes procesadores de la arquitectura. La sección del código que está diseñado para funcionar en paralelo está marcada con una directiva de preprocesamiento que creará los *threads* antes de que la sección se ejecute. Cada *thread* tendrá un identificador (id) que será obtenido a partir de una función (omp_get_thread_num() en C / C++) y después de la ejecución paralela, los *thread* se unirán de nuevo en su ejecución sobre el *thread* maestro, el cual continuará con la ejecución del programa.

Por defecto, cada hilo ejecutará una sección paralela de código independiente pero se pueden declarar secciones de "trabajo compartido" para dividir una tarea entre los hilos, de manera que cada hilo ejecute parte del código asignado. De esta forma es posible tener en un programa OpenMP un paralelismo de datos y un paralelismo de tareas conviviendo conjuntamente. Los principales elementos de OpenMP son las sentencias para la creación de *threads*, la distribución de carga de trabajo, la gestión de datos de entorno, la sincronización de *threads* y las rutinas a nivel usuario. OpenMP utiliza en C/C++ las directivas de preprocesamiento conocidas como *pragma* (#pragma omp <resto del pragma>) para diferentes contrucciones.

Por ejemplo, *omp parallel* se utiliza para crear *threads* adicionales que ejecutarán el trabajo indicado en la sentencia paralela donde el proceso original es el *thread* maestro (id=0). Por ejemplo, el conocido programa que imprime "Hello, world" utilizando OpenMP y *multithreads* es:

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

```
int main(int argc, char* argv[]){
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;}
```

Para especificar *work-sharing constructs* se utiliza:

- *omp for* u *omp do*: reparte las iteraciones de un lazo en múltiples *threads*.
- *sections*: asigna bloques de código independientes consecutivos a diferentes *threads*.
- *single*: especifica que el bloque de código será ejecutado por un solo *thread* con una sincronización (*barrier*) al final del mismo.
- *master*: similar a *single*, pero el código del bloque será ejecutado por el *thread* maestro y no hay *barrier* implicado al final.

Por ejemplo, para inicializar los valores de un *array* en paralelo utilizando *threads* para hacer una porción del trabajo:

```
#define N 100000
int main(int argc, char *argv[]) {
    int i, a[N];
    #pragma omp parallel for
    for (i=0;i<N;i++) a[i]= 2*i;
    return 0;
    }
```

Ya que OpenMP es un modelo de memoria compartida, muchas variables en el código son visibles a todos los *threads* por defecto. Pero a veces es necesario tener variables privadas y pasar valores entre bloques secuenciales del código

y bloques paralelos, por lo cual, es necesario definir atributos a los datos *(data clauses)* para permitir diferentes situaciones:

- *shared*: los datos en la región paralela son compartidos y accesibles por todos los *threads* simultáneamente.

- *private*: los datos en la región paralela son privados para cada *thread* y cada uno tiene una copia de ellos sobre una variable temporal.

- *default*: permite al programador definir cómo serán los datos dentro de la región paralela (*shared*, *private*, o *none)*.

Otro aspecto interesante de OpenMP son las directivas de sincronización:

- *critical section*: el código enmarcado será ejecutado por *threads* pero sólo uno por vez (no habrá ejecución simultánea) manteniendo la exclusión mutua.

- *atomic*: similar a *critical section*, pero avisa al compilador de que use instrucciones hardware especiales de sincronización para obtener mejores prestaciones.

- *ordered*: el bloque es ejecutado en el orden como si de un programa secuencial se tratara.

- *barrier*: cada *thread* espera que los restantes hayan acabado su ejecución (implica sincronización de todos los *threads* al final del código.

- *nowait*: especifica que los *threads* que terminen el trabjo asignado pueden continuar.

Además OpenMP provee de sentencias para la planificación (*scheduling*) del tipo *schedule(type, chunk)* (donde el tipo puede ser *static, dynamic o guided*) o proporciona control sobre la sentencia *if*, que permitirá definir si se paraleliza o no en función de si la expresión es verdadera o no. También OpenMP proporciona un conjunto de funciones librería, como por ejemplo:

- omp_set_num_threads: define el número de *threads* a usar el la siguiente región paralela.
- omp_get_num_threads: obtiene el número de *threads* que se están usando en una región paralela.
- omp_get_max_threads: obtiene la máxima cantidad posible de *threads*.
- omp_get_thread_num: devuelve el número del *thread*.
- omp_get_num_procs: devuelve el máximo número de procesadores que se pueden asignar al programa.

- omp_in_parallel: Devuelve valor distinto de cero si se ejecuta dentro de una región paralela.

Veremos a continuación algunos ejemplos simples:

```
/* Programa simple multihreading con OpenMP */
#include <omp.h>
int main() {
    int iam =0, np = 1;
    #pragma omp parallel private(iam, np)
    {
    #if defined (_OPENMP)
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
    #endif
    printf("Hello from thread %d out of %d \n", iam, np);
    }
}
```

```
/* Programa simple de integración con OpenMP */
main() {
    double local, pi=0.0, w; long i;
    w = 1.0 / N;
    #pragma omp parallel private(i, local) {
    #pragma omp single
    pi = 0.0;
    #pragma omp for reduction (+: pi)
    for (i = 0; i < N; i++) {
        local = (i + 0.5)*w;
        pi = pi + 4.0/(1.0 + local*local); }
    }
}
```

```
/* Programa simple de reducción con OpenMP */
#include <omp.h>
#define NUM_THREADS 2
void main () {
    int i;
    double ZZ, func(), res=0.0;
    omp_set_num_threads(NUM_THREADS)
    #pragma omp parallel for reduction(+:res) private(ZZ)
    for (i=0; i< 1000; i++) {
        ZZ = func(I);
        res = res + ZZ; }
}
```

5.4. Objetos distribuidos

Los paradigmas basados en objetos han cobrado mucha fuerza en la última década, sobre todo por las ventajas que implican este tipo de paradigmas y la evolución de lenguajes como C++ y Java. Desde el punto de vista de estos paradigmas, el elemento principal es el el objeto que podemos considerar como un "elemento software" y que posee las siguientes características: encapsulamiento (es autocontenido e incluye tanto los datos que éste usa –*atributos*– como los procedimientos –*métodos*– que actúan sobre los mismos) y herencia (un objeto puede heredar de alguien un conjunto de atributos/métodos y ampliarlos).

Cuando se utiliza programación orientada a objetos, se definen *clases* (que definen objetos genéricos) y la forma como los objetos interactúan entre ellos. Al ser las clases autocontenidas, mejora notablemente el mantenimiento de la

aplicación, la depuración y permite en forma muy fácil la reutilización de clases para otros programas. Por ejemplo, una clase en Java (*textomovimiento.java* y la clase compilada *textomovimiento.class*) sería (muestra una ventana con texto en movimiento):

```
import java.awt.Graphics;
import java.awt.Font;

public class textomovimiento extends java.applet.Applet implements Runnable {
    char separated[]; String s = null; Thread killme = null;
    int i; int x_coord = 0, y_coord = 0;
    String num; int speed=35; int counter =0;
    boolean threadSuspended = false;

    public void init() {
        resize(550,50);
        setFont(new Font("Arial",Font.BOLD,40));
        s = "Feliz Programación Distribuida!!";
        separated = new char [s.length()];
        s.getChars(0,s.length(),separated,0); }

    public void start() {
        if(killme == null) {
        killme = new Thread(this);
        killme.start(); }
        }

    public void stop() { killme = null; }

    public void run() {
        while (killme != null) {
        try {Thread.sleep(100);} catch (InterruptedException e){}
        repaint(); }
        killme = null; }

    public void paint(Graphics g) {
        for(i=0;i<s.length();i++) {
            x_coord = (int) (Math.random()*10+15*i);
            y_coord = (int) (Math.random()*10+36);
            g.drawChars(separated, i,1,x_coord,y_coord); }
        }

    public boolean mouseDown(java.awt.Event evt, int x, int y) {
        if (threadSuspended) { killme.resume(); }
        else { killme.suspend(); }
        threadSuspended = !threadSuspended;
        return true; }
}
```

Esta clase es heredera de la clase java.applet.Applet e implementa una interfaz *Runnable*, ya que está escrita con *threads* y hecha para ser ejecutada dentro de un navegador como *applet*. Los métodos start, stop, run y paint serán los métodos que llamará la máquina virtual de Java del navegador para inicializar, poner en marcha, parar el *thread* y repintar la ventana respectivamente. El método *mouseDown* permite suspender/reiniciar el *thread* con el ratón. La página texto.html (por ejemplo) que ejecutará este programa Java será:

```
    <html>
    <head> <title>Texto en movimiento</title></head>
    <body>
    <h1>Ejemplo de clases en Java</h1><hr>
```

```
<applet code="textomovimiento.class" height="50" width="550"></applet>
</body>
</html>
```

Una de las evoluciones del paradigma es permitir que los objetos se ejecuten en un sistema remoto y devuelvan el resultado a la aplicación que lo ha invocado. **Corba** es una arquitectura desarrollada por Object Management Group (un consorcio muy grande de investigadores e industrias) que han definido un estándar (*common objet request broker architecture*), que define todos los aspectos necesarios para trabajar con aplicaciones y objetos distribuidos. Las grandes ventajas de este tipo de paradigma es que la máquina donde se ejecuta la aplicación sólo es un contenedor y las prestaciones vendrán dadas por la máquina remota donde se ejecutan los objetos remotos. El cliente sólo pide una información a un objeto y no se preocupa de cómo está implementado. Corba, es decir el ORB, ante la solicitud, podrá en marcha el servidor y atenderá la petición y retornará el resultado a quien lo solicitó. Algunos autores consideran la ORB como la evolución de las RPC. Obviamente, por todo ello se debe pagar un precio: Corba asegura el funcionamiento de los objetos distribuidos en la red pero la tarea del programador se complica al utilizarlo, es decir, Corba no es fácil de utilizar pero garantiza el trabajo con objetos distribuidos.

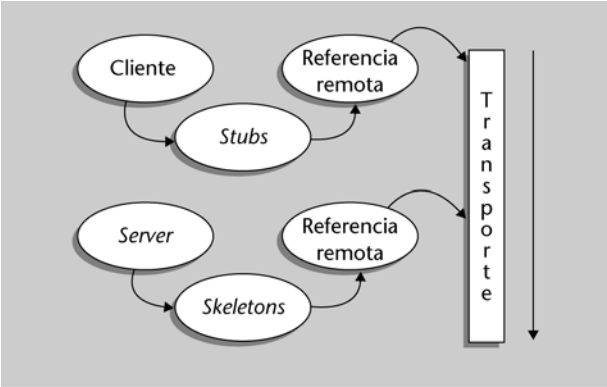
Java remote method invocation (JRMI) es un entorno de trabajo simple y potente para trabajar con objetos distribuidos y no presenta la complejidad de trabajo de algunas de las implementaciones de Corba. Los objetos con RMI pueden ser diseñados muy fácilmente y ser puestos en la red permitiendo de forma rápida y sencilla DOP (*distributed object programming*). El único inconveniente de RMI es que los objetos y programas que los invocan deben estar escritos en Java.

La idea de programación distribuida es muy simple y efectiva, ya que distribuimos el trabajo (*farm*) a otras aplicaciones en otras máquinas y recibimos el resultado. RMI permite distribuir el trabajo sobre otros objetos en la red y no sólo utilizar paralelismo local (*threads*), sino paralelismo global: en otras máquinas, otros SO, otro HW, pero con igual lenguaje de programación.

¿Cómo trabaja RMI? Cuando un cliente llama a un servidor RMI, varias capas de software se ponen en juego. Para el programador, las más importantes son las de *stub* –para el código del cliente– y *skeleton* –para el código del *server*– (código Java que se deberá rellenar para poder comunicarnos con las otras capas), que derivarán de las clases RMI. En primer lugar de deberá inicializar la RRL (*remote reference layer*), que indicará dónde estará el objeto (máquina o red) y cómo se utilizará, y después se deberá configurar la TL (*transport layer*), que transformará el código RMI en TCP/IP (o cualquier otro protocolo de red). Dado que RMI soporta serialización de objetos (*object serialization*), no importa cómo sean de complicados los objetos pasados como parámetros, ya que éstos son convertidos en *streams* cuando

viajan por la red y fácilmente reconvertidos en el objeto original cuando llegan al otro extremo. Las capas que intervienen en la arquitectura RMI son las que muestra la figura siguiente:

Figura 15



- SSL (*stub/skeleton layer*): se activa cuando el cliente invoca el servidor a través de la API con la cual trabaja el programador (que es heredado de las clases RMI). Es decir, todo lo referente a la comunicación, duplicación de objetos, seguridad, etc., queda oculto por estas capas.

- RRL (*remote reference layer*): maneja la translación entre SSL y las llamadas de la arquitectura nativa estableciendo un puente entre el código de la aplicación y la red de comunicaciones. También es la responsable de controlar las excepciones.

- TL (*transport layer*): rutinas no accesibles al usuario, responsables de iniciar las comunicaciones, mantenerlas, controlar los errores, escuchar por conexiones y cerrarlas.

Para crear un sistema distribuido basado en objetos distribuidos, deberemos organizar la aplicación en un objeto cliente y otro objeto servidor. La metodología para crear cliente RMI se basa en tres pasos: a) obtenemos el objeto desde el NS (*Naming Service*), b) procesamos el objeto (objeto disponible para su utilización), c) utilizamos el objeto.

A continuación veremos un ejemplo simple (*Stats*, obtener datos estadísticos de un objeto remoto) para crear un objeto cliente que llamará a objetos remotos para actualizar una suma. En este ejemplo se utilizará para el servidor la clase UnicastRemoteObject, que es una implementación de la clase RemoteObject, de esta forma, si se quieren crear dos versiones de la aplicación (una con objetos remotos y otra con objetos locales) simplemente se debe cambiar la herencia de la clase. Un parte importante de la arquitctura RMI es el *Naming System* la cual permite referirse a los objetos como un *string*. Para ello es necesario poner en marcha el registro cuya función es asegurar que un objeto está disponible para su uso. El registro debe ponerse en marcha, tanto en la máquina local como en la remota, ejecutando la aplicación Java: *java sun.rmi.registry.RegistryImpl*.

La interfaz será:

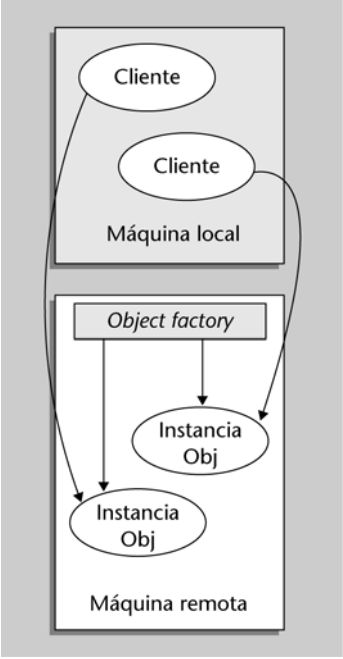
```
// Archivo: StatsInterface.java Que hace: Interfaz básica para Stats
import java.rmi.*;

public interface StatsInterface extends Remote {
String getStats( String teamName) throws RemoteException;
}

// Archivo: StatsClient.java Que hace: la GUI del cliente para acceder a stats
import java.awt.*; // Las clases Java
import java.applet.*;
import java.rmi.registry.*; // Las clases Java RMI
import java.rmi.*;
import java.rmi.server.StubSecurityManager;

public class StatsClient {
public static void main( String args[]) {
    Remote statsRemoteObj = null; // el objeto remoto
    StatsInterface statsInterface = null;
    System.setSecurityManager(new StubSecurityManager()); // el RMI Security manager
    System.out.println("Security Manager...");
      try {  // obtenemos el objeto remoto del Registry
      statsRemoteObj = Naming.lookup("STATS"); }
    catch(Exception exc) {
      System.out.println("Error Naming - " + exc.toString());
      System.exit(1);
}
    System.out.println("No existe el servidor en Naming...");
    try { // obtenemos la referencia al servidor
      statsInterface = (StatsInterface) statsRemoteObj; } // Cast al mi objeto
    catch(Exception exc) {
      System.out.println("Error con el cast - " + exc.toString());
      System.exit(1); }
    System.out.println("Servidor listo...");
    try {
      String stats = statsInterface.getStats("40x2");
      System.out.println("Stats: [" + stats + "]");}
catch(Exception exc) {
      System.out.println("Error durante la llamada - " +
         exc.toString()); }

    } // Fin main
} // Fin Clase StatsClient

// Archivo: StatsServer.java Qué hace: Es el servidor de objetos remotos.
import java.rmi.*;
import java.rmi.server.*;

public class StatsServer extends UnicastRemoteObject implements StatsInterface {
     StatsServer() throws RemoteException { // Constructor
         super(); }

    public String getStats( String teamName ) throws RemoteException {
         return "es 80"; } // la función que estamos implementando

    public static void main(String args[] ) { // Servidor -standalone-
       try {
         System.setSecurityManager(new RMISecurityManager()); // SM
         StatsServer statsServer = new StatsServer(); // Instancia del objeto
         Naming.rebind("STATS", statsServer); } // Publico
       catch(Exception exc){
         System.out.println("Error en main - " + exc.toString()); }
    } // Fin main
} // Fin Clase StatsServer
```

Para compilar haremos:

```
    javac -g -d {CLASS_DIR} -classpath {CLASS_PATH} *.java
On {CLASS_DIR} se ha fde reemplazar con el directorio donde estarán las
clases, y {CLASS_PATH} con el directorio donde están las clases de Java.
    rmic StatsServer (RMI compiler: generará los S&S, códigos de
inicialización de objetos).
El último paso es publicar la aplicación en el registro, momento en que
estará disponible el objeto para los clientes.
En la banda del servidor, se hará lo siguiente:
    java sun.rmi.registry.RegistryImpl
    java StatsServer
En la banda del cliente se hará lo siguiente:
    javasun.rmi.registry.RegistryImpl
    java StatsClient
```

Un aspecto interesante de RMI es que permite crear servidores dinámicos para hacer instancias de objetos diferentes para cada llamada. Esto se realiza a través de "factoría de objetos", donde el esquema que se utiliza es como el indicado en la figura siguiente (consultar la bibliografía para ampliar este contenido):

Figura 16



Otro aspecto innovador de los objetos distribuidos es el mecanismo de Call-Backs, que permite al servidor de objetos remotos modificar datos del objeto local y evitar así que la llamada local interrogue continuamente al objeto remoto por una actualización de datos que en la mayoría de los casos puede ser innecesaria (es equivalente a un sistema de interrupciones de un programa: se activa cuando hay un acción para realizar). Consultar la bibliografía para ampliar este contenido.

Microsoft ha desarrollado ideas equivalentes de objetos distribuidos que se encuentran en la arquitectura DCOM (*distributed component object model*). Esta arquitectura permite la interoperabilidad y la vinculación con las capas del sistema operativo subyacente generando aplicaciones eficientes en sistemas

Windows. El programador dispone de un conjunto de clases y servicios que permiten construir aplicaciones distribuidas basadas únicamente en la arquitectura .Net e integrar objetos locales, remotos, componentes y servicios web en diferentes capas y con interfaces (bien) definidas.

5.5. Modelos de componentes

Un modelo de programación que complementa a los anteriores son los basados en modelos de componentes (*component model* CM). Un modelo de componentes se basa en que varias partes o elementos trabajan conjuntamente en una aplicación. Java Beans (Sun), Active X (Microsoft) y OpenDoc (Apple), entre otros, que soportan la idea de modelos de componentes. La principal ventaja de los modelos de componentes es que permiten la generación de componentes (elementos software consistentes en uno o varios objetos) "reutilizables" haciendo rentable el esfuerzo empleado en el desarrollo del software, encapsulando los elementos y habilitando que otros puedan utilizarlo.

La idea se basa en un elemento (o un *bean*) que es un componente que encapsula o envuelve otros objetos. Este componente posee una interfaz bien definida para el grupo de objetos que integra y que permite la interacción con los objetos agrupados, haciendo de esta forma que la reutilización esté garantizada. Un ejemplo de un modelo de componentes es un automóvil. Todas las partes deben estar en su sitio para que el coche funcione como debe, es decir, éste requiere un CM para que funcione correctamente. Sin CM el coche sería un conjunto de piezas desorganizadas que individualmente no podrían realizar ninguna tarea útil (o por lo menos con el fin que fueron concebidas). Los modelos de componentes no son necesariamente un método de programación en red (*network programming*), sino que proveen de un medio para agrupar componentes en una red sobre el mismo paraguas.

Si por ejemplo se utiliza la tecnología Java (Java Beans –JB–) y dado que los *beans* están escritos en Java, son portables y se basan en el principio *escriba-una-vez-utilice-para-siempre* permitiendo crear en forma muy fácil aplicaciones distribuidas. Es por ello por lo que no hay restricciones sobre cómo son los componentes de un *bean*, permitiendo la distribución de *applets* y aplicaciones formadas por ellos. Además, los Java Beans no interfieren con los mecanismos de comunicación (IDL, RMI), sino que han sido diseñados para convivir con ellos. Por ejemplo, si deseamos crear un conjunto de *applets* sobre una página web, encontraremos una de las limitaciones de los *applets*: no se pueden comunicar entre ellos (un evento sobre un *applet* no será reconocido por otro). Java Beans permite crear un contenedor que contiene dentro *applets* y aplicaciones que podrán comunicarse libremente mientras estén dentro del mismo recipiente (un ejemplo gráfico son los clásicos Tupperware donde los recipientes más grandes contienen los más pequeños y todos están dentro del de mayor tamaño).

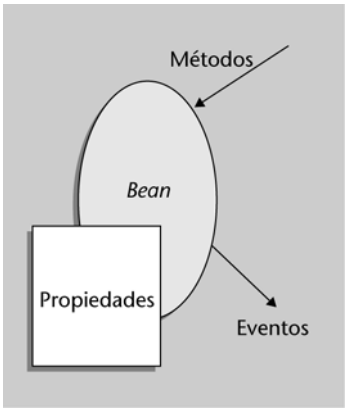
Una de las principales ventajas de JB con respecto al modelo MS ActiveX (MSAX) es la flexibilidad con independencia de plataforma (es considerada como un *open solution*). Los MS AX depende del sistema operativo y del navegador Internet Explorer (si bien hay algunos *plugins* experimentales para otros navegadores) pero el servidor siempre debe estar sobre una plataforma Windows y es por ello por lo que algunos autores difieren de la clasificación de Microsoft (*open solution*) diciendo que es un Microsoft-SS (*specific solution*) por sus dependencias respecto a IE y Windows.

Un aspecto crucial es la seguridad de estos modelos, ya que se ejecutarán objetos encapsulados sobre la máquina local. Dado que MS-AX está integrado por Windows, es más fácil enmascarar código *malware* u otro programa perjudicial para la máquina local dentro de ellos. Para solucionar este problema, Microsoft propone una tecnología basada en *Authenticode,* que permite que el usuario preseleccione de qué sitios aceptará AX, pero con esto la seguridad no está garantizada. Por el contrario, JB está ligado a un *applet* y permanece estrictamente dentro Java "*sandbox*", que previene el acceso a cualquier archivo dentro de la máquina local y por lo cual se considera como una tecnología segura.

5.5.1. Java Beans

Cuando se utiliza JB, se dispone de un conjunto de ventajas: igual modelo de seguridad, igual modelo de interacción e igual modelo de eventos, ya que JB no contradice en nada el espíritu de Java. En JB cada *bean* tiene un conjunto de propiedades y puede ser invocado por varios métodos disparando eventos sobre otros *beans,* por lo cual, simplemente publicando la API, un determinado *bean* puede mostrar a otro sus propiedades y métodos y disparar eventos sobre las API publicadas de otros *beans*.

Figura 17



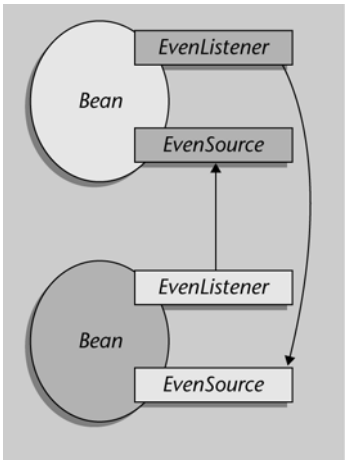
La pieza fundamental de un diseño distribuido efectivo es decidir qué es remoto y qué es local. Para que este objetivo no sea complicado, es deseable crear diseños utilizando un módulo de red que reciba la comunicación y la traslade a los módulos que la utilizan. Esta forma permite facilidad en el cambio y adaptación dinámica a las circunstancias de carga.

Entre las principales características de JB, podemos mencionar:

- Persistencia: se puede salvar el estado de sus propiedades entre ejecuciones, es decir, sus valores no son inicializados.
- Eventos: JB provee un mecanismo de notificación de eventos que permite propagar un evento a otros *beans* en su CM.
- Propiedades: dado que son clases, se pueden heredar y crear *beans* basados en otros *beans*.
- Potencia el diseño OO: en lugar de publicar librerías de clases, se publican objetos.

La idea base de diseño es crear cada parte de su aplicación como se haría normalmente, donde cada *applet*, documento, componente de la aplicación *beans* podría ser desarrollado, depurado con antelación. A partir de este momento, se debe crear un *bean* que encapsule todos los componentes y que incorpore dos objetos críticos para la gestión del flujo y almacenamiento de la información. Los eventos serán intercambiados por los objetos *EventListener* y *EventSource*. El objeto *EventListener* se crea para mirar por cierto tipo de eventos dentro de la aplicación y donde cada *bean* creará un *listener* si quiere recibir eventos. Un *bean* también puede crear un *EventSource* para crear y publicar eventos para otros *beans* dentro de la aplicación.

Figura 18



A continuación se mostrará como ejemplo el código de un tirador de penales y un portero, donde cada uno es un objeto independiente que está agrupado en *beans*.

```
//Archivo JugadorListenerI.java
public interface JugadorListener extends EventLister {
    public void throwJugador ( String jugador); }
// Nuestro juego debe ser configurado para escuchar eventos disparados
por el jugador.
// Para ello creamos una interfaz para los "lanzamientos" como una
extensión de EventLister y agregamos un // método (throwJugador) que será
implementado más adelante en el FutbolGame para conectar ambas clases.
```

```
//Archivo jugador.java
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;
import java.io.Serializable;
import java.beans.*;

public class Jugador implements Serializable {
    private Vector miListeners;
    private Button pelotaRapida;
    private Button pelotaEfecto;
    private Button pelotaFuerte;
    public Jugador() { // constructor
        setLayout(new GridLayout(4, 1)); setBackground(Color.lightGray);
        pelotaRapida = new Button("Pelota Rápida"); add(pelotaRapida);
        pelotaEfecto = new Button("Pelota con Efecto"); add(pelotaEfecto);
        pelotaFuerte = new Button("Pelota Fuerte"); add(pelotaFuerte);
        miListeners = new Vector(); } // inicializamos nuestro listener }

    public void addListener( JugadorListener listener ) { // Añadimos un listener
    miListeners.addElement(listener); }
    public void removeListener(JugadorListener listener ) { // Borramos un listener
    miListeners.removeElement(listener);  }

    public boolean action( Event evt, Object obj)   { // algo pasa
        if(evt.target instanceof Button)  { // Botón pulsado
            String p = new String( (String) obj);
            // Creo una acción basada en el botón que se pulse
            for ( int x = 0; x < miListeners.size(); x++) {
            // Vamos al vector y obtenemos el evento
                   JugadorListener listener = (JugadorListener)
                   milisteners.elementAt(x);
                   listener.throwJugador(p); }
        }
    }
} // Clase Jugador

public class Portero extends Panel    { // No necesita implementar
código bean,
    // El Jugador lanzará los eventos sobre el portero como si fueran
eventos normales de botones,
    // y el portero responderá a ellos
    TextArea jugadorArea;

    Portero ( ) {
        setLayout(new GridLayout(1, 1)); setBackground(Color.lightGray);
        jugadorArea = new TextArea ();
        add (jugadorArea); }
    public void AtajoTiro ( String tiro ) {
        jugadorArea.addText ("Y el Jugador es : " + tiro); }
} // Clase portero

//Archivo FutbolGame.java
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.beans.*;
public class FutbolGame extends Applets implements JugadorListener {
    Jugador jugador; Portero portero;
    FutbolGame (String fieldName ) {
        setLayout(new GridLayout(1, 2)); resize(500, 400); // inicializo la GUI
        jugador = new Jugador("Jugador");
        add(jugador); // creo los Beans
        portero = new Portero("Portero");
        add(portero);
        jugador.addListener(this); // El jugador dispara eventos y el portero los recibe
        } // Añadimos el game como un listener para el jugador
    public void throwJugador (String newJugador) {
        // Indicamos al portero que ataje mi lanzamiento
          Portero.AtajoTiro( newJugador);
    }
}
```

5.6. *Web Services*

La W3C define *web services* (WS) como el software de sistema diseñado para soportar interacciones máquina-máquina con interoperabilidad sobre la red. Los WS son como la API web que puede ser accedida sobre una red, tal como Internet, y ejecuta sobre un sistema remoto los servicios requeridos. La definición de la W3C implica diferentes sistemas pero la utilización del término se refiere a cliente y servidores que se comunican utilizando mensajes XML y que siguen el estándar SOAP. También se pueden encontrar operaciones interpretadas por el servidor y escritas en *Web Services Description Language* (WSDL) y este último no es un requisito del un *SOAP endpoint*, pero sí que es un prerrequisito para generación de código automático del lado del cliente en el ámbito SOAP sobre Java o .Net.

Las especificaciones que definen los servicios web son modulares y, en consecuencia, no hay un solo documento que los contenga a todos, sin embargo, entre las especificaciones de base podemos enumerar:

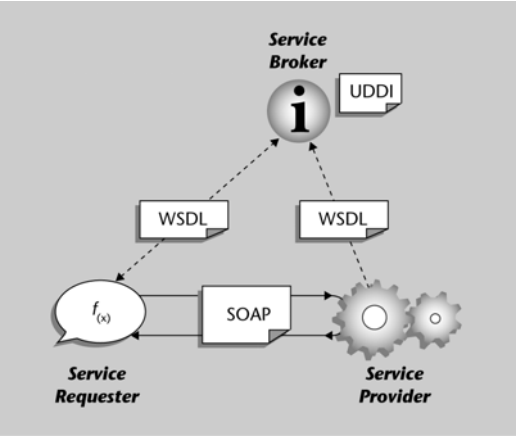
- **SOAP:** basada en XML y extensible, con formato de mensaje sobre "referencias" de los protocolos subyacentes. Los protocolos principales son HTTP y HTTPS, pero hay implementaciones para otros como SMTP y XMPP.

- ***Web Services Description Language* (WSDL):** un formato XML que permite las interfaces de servicio que se describen junto con los detalles de sus enlaces a protocolos específicos. Se suele utilizar para generar código de cliente y servidor y para la configuración.

- ***Universal Description Discovery Integration* (UDDI):** es un protocolo para la publicación y el descubrimiento de metadatos acerca de los servicios web, que permite a las aplicaciones encontrar a éstos, ya sea en tiempo de diseño o de ejecución.

La mayoría de estas especificaciones básicas se han generado en la W3C, incluyendo XML, SOAP, WSDL; UDDI proviene de la organización OASIS. Para mejorar la interoperabilidad de los WS, la Web Services Interoperability Organization (WS-I) utiliza *profiles*. La arquitectura de los web services de la W3C se muestra en la figura siguiente:

Figura 19



Algunas especificaciones han sido desarrolladas (o están en desarrollo) para extender las capacidades del WS. Éstas se conocen generalmente como **WS-***, una lista de las más interesantes podría ser:

- **WS-Security:** define cómo utilizar **XML Encryption y XML Signature** en **SOAP** para realizar intercambio seguro de mensajes como alternativa o extensión a la utilización de HTTPS.
- **WS-Reliability:** es un protocolo estándar de **OASIS** para el intercambio fiable de mensajes entre dos WS.
- **WS-Addressing**: una forma de describir la dirección del recipiente (y el remitente) de un mensaje dentro del mensaje SOAP.
- **WS-Transaction:** una forma de manejar las transacciones.

Los WS son un conjunto de herramientas que pueden ser utilizadas de diferentes formas. Los tres estilos más comunes de uso son RPC, SOA y REST.

a) WS RPC: presentan una interfaz a función (o método) remota familiar a muchos desarrolladores. Normalmente, la unidad básica de una WS RPC es una operación WSDL.

Los primeros WS se centraron en RPC, siendo este método ampliamente desplegado y apoyado; sin embargo, a veces este método ha sido criticado por no ser desacoplados respecto al servicio y a la función en un lenguaje específico.

b) SOA (arquitectura orientada a servicios): los WS también se puede utilizar para implementar una arquitectura de acuerdo con los conceptos de los servicios orientados hacia la arquitectura (SOA), en donde la unidad básica de la comunicación es un mensaje, en vez de una operación. Algunos autores lo definen como *mensaje orientado* a los servicios.

c) REST: WS RESTful es el intento de emular los protocolos HTTP y similares al restringir el interfaz a un conjunto de operaciones bien conocidas (por ejemplo, GET, PUT, DELETE). Aquí, la atención se centra en la interacción con los recursos en lugar de los mensajes o de las operaciones. RESTful puede utilizar WSDL para describir mensajes SOAP sobre HTTP, que define las operaciones, o se puede implementar como una abstracción puramente en la parte superior de SOAP (por ejemplo, WS - Transferencia).

5.6.1. Un ejemplo de programación con JAX-WS

Un objetivo clave de JAX-WS es simplificar el desarrollo de servicios Web Java. En el modelo de programación de JAX, la definición de un servicio web puede comenzar desde cualquier clase Java o documento WSDL. También es posible comenzar con un WSDL y una clase Java, y definir un servicio web a través de personalizaciones. El modelo de programación consta de los siguientes pasos:

1) Definir un servicio de ejecución de Java.

2) Generar el servicio web, entre ellos un fichero WSDL, mediante la aplicación de las sentencias al fichero fuente. Este paso se realiza con la ayuda de la herramienta apt, que forma parte del JDK.

3) Empaquetado de los archivos creado en el paso anterior en un archivo war junto con uno o más descriptores. Éste se considera una creación *raw* del archivo war a diferencia del siguiente que será *cooked*.

4) Generar el archivo war *cooked,* que incluirá clases adicionales y los datos de configuración. Para esta tarea se utilizará la herramienta wsdeploy, que es parte del paquete JAX.

JAX no exige que un servicio web implemente una interfaz y la definición de los servicios web puede ser tan sencilla como escribir esta clase Java:

```
package server;
import javax.jws.WebService;
@WebService
public class HelloImpl {
    /*** @param name
        @return Say hello to the person. */
    public String sayHello(String name) { return "Hello, " + name + "!";}
    }
```

Lo que hace que esta clase sea un servicio web es la anotación *@WebService*. Esta anotación es una de las definidas en la web de Servicios-Metadatos para la especificación de la plataforma Java y marca esta clase Java como una aplicación de un servicio web. Se debe tener en cuenta que esta clase no implementa una interfaz, y que el método sayHello() no declara una RemoteException. Una clase de servicio web es una implementación del método con parámetros y tipos de retorno que debe ser compatible con JAXB 2.0. El JAX-WS se basa en la aplicación de procesamiento apt y se aplica a las fuentes Java para crear archivos de código adicional. El próximo paso es ejecutar apt sobre el código Java, lo que genera los siguientes archivos:

```
    HelloServiceImpl.wsdl
    schema1.xsd
    classes/server/HelloImpl.class
    classes/server/jaxrpc/SayHello.class
    classes/server/jaxrpc/SayHelloResponse.class
    classes/server/jaxrpc/SayHello.java
    classes/server/jaxrpc/SayHelloResponse.java
```

El archivo fundamental es el wsdl, que genera la descripción del servicio a partir de la clase anterior:

```
<?xml version="1.0" encoding="UTF-8"?>
     <definitions
     name="HelloImplService"
     targetNamespace="http://server/jaxrpc"
     xmlns:tns="http://server/jaxrpc"
     xmlns="http://schemas.xmlsoap.org/wsdl/"
```

```
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

<types>
 <xsd:schema>
  <xsd:import namespace="http://server/jaxrpc"
     schemaLocation="schema1.xsd"/>
</xsd:schema>
</types>

<message name="sayHello">
     <part name="parameters" element="tns:sayHello"/>
</message>
<message name="sayHelloResponse">
  <part name="result" element="tns:sayHelloResponse"/>
</message>

<portType name="HelloImpl">
   <operation name="sayHello">
    <input message="tns:sayHello"/>
    <output message="tns:sayHelloResponse"/>
    </operation>
</portType>

<binding name="HelloImplBinding" type="tns:HelloImpl">
   <soap:binding
  transport="http://schemas.xmlsoap.org/soap/http"
   style="document"/>

<operation name="sayHello">
   <soap:operation soapAction=""/>
   <input>
     <soap:body use="literal"/>
    </input>
    <output>
     <soap:body use="literal"/>
      </output>
   </operation>

</binding>
<service name="HelloImplService">
   <port name="HelloImplPort" binding="tns:HelloImplBinding">
     <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
    </port>
</service>
</definitions>
```

La definición WSDL importa el archivo de esquema schema1.xsd. Este esquema define dos tipos complejos de la WSDL (sayHelloResponse y sayHello) y su contenido es el siguiente:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0"
   targetNamespace="http://server/jaxrpc"
     xmlns:xs="http://www.w3.org/2001/XMLSchema">

   <xs:element name="sayHelloResponse"
      type="ns1:sayHelloResponse"
      xmlns:ns1="http://server/jaxrpc"/>

 <xs:complexType name="sayHelloResponse">
   <xs:sequence>
     <xs:element name="return" type="xs:string" minOccurs="0"/>
   </xs:sequence>
</xs:complexType>

<xs:element name="sayHello"
    type="ns2:sayHello"
    xmlns:ns2="http://server/jaxrpc"/>
```

```
    <xs:complexType name="sayHello">
     <xs:sequence>
         <xs:element name="name" type="xs:string" minOccurs="0"/>
     </xs:sequence>
    </xs:complexType>

    </xs:schema>
```

Las dos clases de Java definen los *beans* JAXB que se utilizan para serializar y deserializar (*marshal*, *unmarshal*) los mensajes del servicio web. *Apt* compila esas fuentes, junto con la definición de servicio, y genera los archivos de clase.

```
package server.jaxrpc;
    import javax.xml.bind.annotation.*;
    import javax.xml.rpc.ParameterIndex;
    @XmlRootElement(name="sayHelloResponse",
           namespace="http://server/jaxrpc")
    @XmlAccessorType(AccessType.FIELD)
    @XmlType(name="sayHelloResponse",
        namespace="http://server/jaxrpc",
        propOrder={"_return"})
public class SayHelloResponse {
      @XmlElement(namespace="", name="return")
      @ParameterIndex(value=-1)
      public java.lang.String _return;
      public SayHelloResponse(){}
    }

package server.jaxrpc;
    import javax.xml.bind.annotation.*;
    import javax.xml.rpc.ParameterIndex;
    @XmlRootElement(name="sayHello",
        namespace="http://server/jaxrpc")
    @XmlAccessorType(AccessType.FIELD)
    @XmlType(name="sayHello",
        namespace="http://server/jaxrpc",
        propOrder={"name"})
public class SayHello {
      @XmlElement(namespace="", name="name")
      @ParameterIndex(value=0)
      public java.lang.String name;
      public SayHello(){}
    }
```

A continuación es necesario generar un paquete en un archivo war junto con uno o más descriptores del proyecto. Esta información estará en el archivo jaxrpc-ri.xml y contiene información sobre un servicio web:

```
<?xml version="1.0" encoding="UTF-8"?>
<webServices
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/dd"
    version="1.0"
    targetNamespaceBase="http://artima.com"
    typeNamespaceBase="http://artima.com"
    urlPatternBase="/ws">

    <endpoint
    name="sayhello"
    displayName="A Java Web service"
    interface="server.HelloImpl"
    implementation="server.HelloImpl"
    wsdl="/WEB-INF/HelloImplService.wsdl"/>

    <endpointMapping
    endpointName="sayhello"
    urlPattern="/sayhello"/>
</webServices>
```

Este descriptor define un único punto sayhello, y lo mapea en la URL /sayhello (la interfaz del servicio y la implementación se refieren a la misma clase).

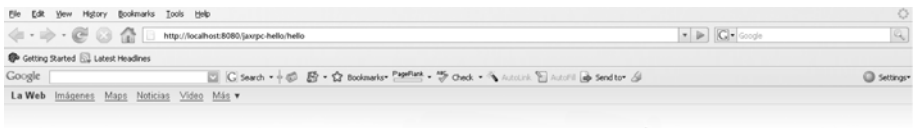
```
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<web-app>
    <display-name>Hello, JAX-RPC 2.0!</display-name>
    <description>A hello, world, Web service.</description>
    <session-config>
    <session-timeout>60</session-timeout>
    </session-config>
</web-app>
```

Crear un archivo *WAR file* es fácil con la herramienta de ant war, donde el archivo de *schema* será colocado en el directorio raíz, así podrá ser descargado por el cliente que necesite acceder a la descripción WSDL. La descripción WSDL será servida por el *servlet* JAX-WS y la estructura de los archivos war será la siguiente:

```
    schema1.xsd
    WEB-INF/web.xml
    WEB-INF/jaxrpc-ri.xml
    WEB-INF/HelloImplService.wsdl
    WEB-INF/classes/server/HelloImpl.class
    WEB-INF/classes/server/jaxrpc/SayHello.class
    WEB-INF/classes/server/jaxrpc/SayHello.java
    WEB-INF/classes/server/jaxrpc/SayHelloResponse.class
    WEB-INF/classes/server/jaxrpc/SayHelloResponse.java
```

El paso final es generar el archivo war definitivo conocido como *cooked war*, lo cual añadirá los descriptores finales y otras clases de soporte utilizando la herramienta *wsdeploy*. *Wsdeploy* lee *jaxrpc-ri.xml* e invoca a la herramienta wscompile para generar el archivo war "*cooked*". Este archivo podrá ser copiado en el directorio *webapps* (contenedor de *servlets*) del servidor web que ya estará listo para utilizarse (todos los archivos jar deberán ser copiados en el directorio de librerías, por ejemplo en Jakarta Tomcat en /shared/lib). Se podrá acceder al servicio mediante http://localhost:8080/jaxrpc-hello/hello donde *localhost* y el puerto deberán ser sustituidos por el nombre del servidor y el puerto donde se atienden las peticiones. El resultado será:

Figura 20



Web Services

Port Name	Status	Information	
sayhello	ACTIVE	Address:	http://localhost:8080/jaxrpc-hello/hello
		WSDL:	http://localhost:8080/jaxrpc-hello/hello?WSDL
		Port QName:	{http://server/jaxrpc}HelloImplPort
		Remote interface:	server.HelloImpl
		Implementation class:	server.HelloImpl
		Model:	http://localhost:8080/jaxrpc-hello/hello?model

La descripción WSDL del servicio web será accesible a través de esta URL (para descargarse el archivo WSDL):

```
http://localhost:8080/jaxrpc-hello/hello?WSDL
```

El cual dará el siguiente contenido:

```
<service name="HelloImplService">
    <port name="HelloImplPort"
        binding="tns:HelloImplBinding">
            <soap:address location="http://localhost:8080/jaxrpc-hello/hello"/>
    </port>
</service>
```

6. Casos de uso: paradigmas y complejidad

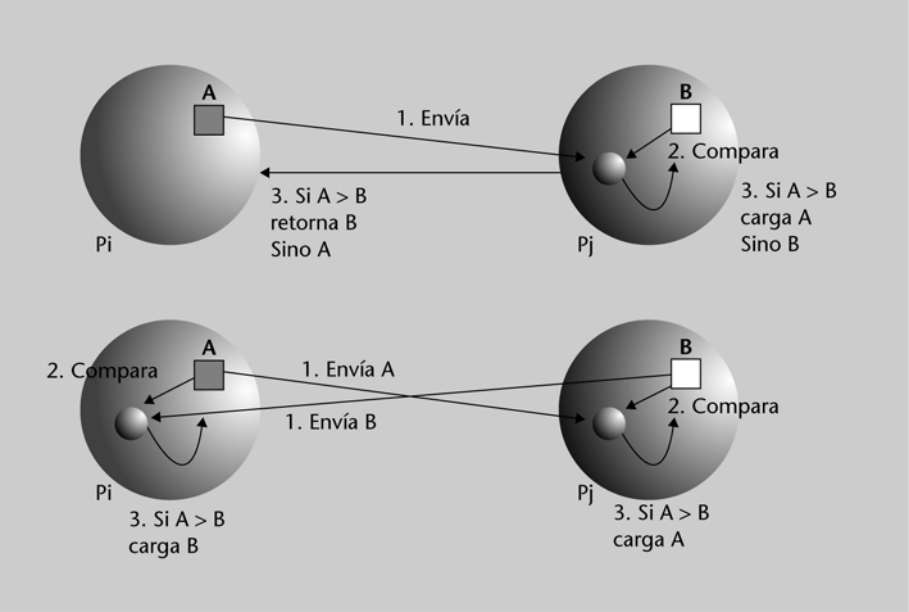
Un aspecto crucial en la programación paralela/distribuida es seleccionar el algoritmo más adecuado para realizar una tarea y no es nada fácil para un programador, ya que se deben tener en cuenta muchas consideraciones, como por ejemplo la arquitectura hardware subyacente (*cores*, memoria, ancho de banda, etc.), portabilidad, eficiencia, complejidad, escalabilidad, prestaciones o distribuciones de datos. Todas estas cuestiones pueden hacer que un algoritmo que funciona bien sobre un paradigma de memoria compartida sea poco eficiente sobre objetos distribuidos, pero a su vez con escalabilidad nula sobre los primeros y por el contrario, con escalabilidad ilimitada si están basados en objetos remotos.

Para analizar estas cuestiones, hemos decidido analizar el problema que se presenta frecuentemente sobre las necesidades de ordenación de datos en forma paralela/distribuida y analizaremos algunas implementaciones de los algoritmos de ordenación mirando cuáles podrían ser los mecanismos de selección del algoritmo/paradigma antes de iniciar la escritura del código. Generalmente, los algoritmos de ordenación se basan en la acción de comparar e intercambiar información o en base a la propia representación de los números. La complejidad (O) es un medida utilizada normalmente para definir cuánto costará realizar la acción propuesta e implica (generalmente) una medida de prestaciones. En los algoritmos de comparar-intercambiar, el límite inferior de la complejidad es O(n*log n), mientras que en los basados en la representación, el límite inferior de la complejidad es O(n). En ordenación secuencial, los mejores algoritmos (*quicksort, mergesort*) tienen complejidad O(n*log n), por lo cual, si tuviéramos *n* procesadores se puede demostrar que O(n*log n)/n=O(log n), que sería lo deseable. A esta progresión la llamaremos *escalabilidad* y se refleja generalmente en un valor conocido como *speedup*. Un *speedup* lineal significa que si el algoritmo secuencial tarda X tiempo en resolver el problema, el algoritmo paralelo o distribuido en *n* procesadores tardará X/n tiempo.

Considerando los algoritmos de ordenación paralelos, deberemos tener en cuenta dos aspectos fundamentales, que son la distribución de datos y cómo se realizan las comparaciones. En el primer caso, se debe diferenciar si todos los elementos a ordenar están en el mismo procesador o si cada procesador contiene un bloque de datos y si se cumple que todos los elementos del procesador P_i son menores que los de Pj si i<j. Es decir, están desordenados en el procesador pero existe una cierta relación global. En cuanto a las comparaciones, se pueden realizar teniendo en cuenta que cada procesador tiene un único elemento o que cada procesador tiene un conjunto de elementos. Las situaciones posibles se muestran a continuación.

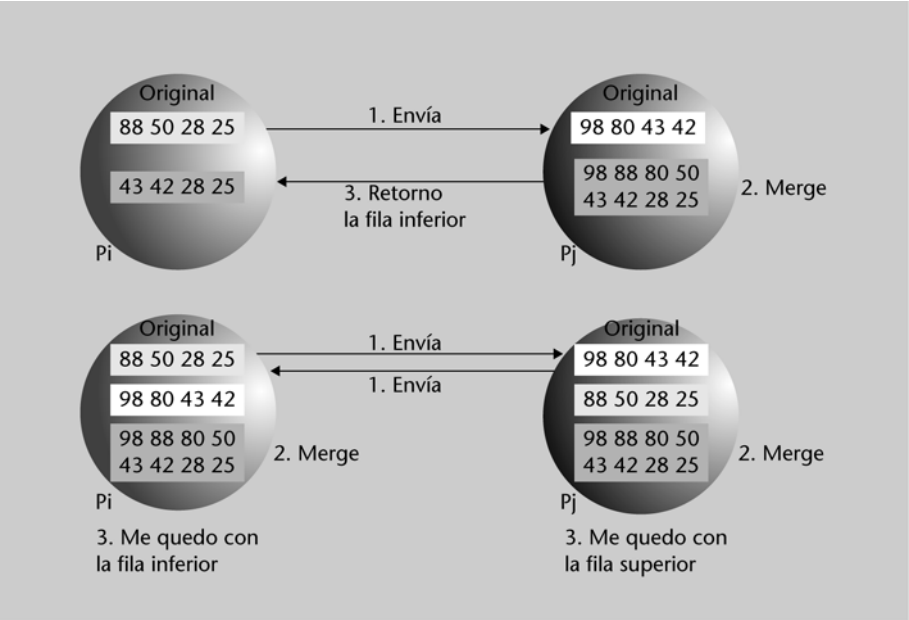
En el primer caso suponemos un elemento por procesador (en el primero sólo Pj realiza la ordenación, mientas que en el segundo Pi y Pj realizan la ordenación) y en comunicaciones, el gasto es el mismo, pero en prestaciones puede ser diferente, ya que en el segundo no es un bloqueo hasta que recibe el dato, sino que es una comparación extra que debe hacer el proceso Pi.

Figura 21



No obstante, este tipo de distribución no implicará incremento de *speedup*, ya que el Pi y Pj deberán bloquearse por el envío de datos. En este caso, todos los paradigmas son posibles pero el más adecuado podría ser MPI y Open MP para Pj (también podría utilizarse una RPC llamada por Pi sobre Pj).

Figura 22



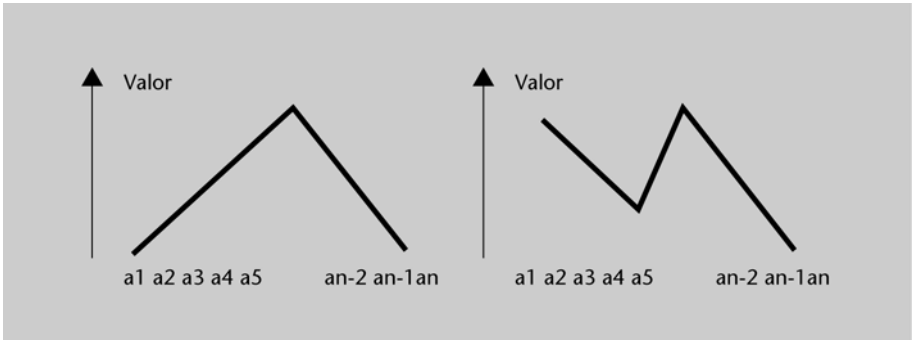
En el segundo caso consideramos más de un elemento por procesador y en la primera figura, sólo Pj realiza las comparaciones mientras que en la segunda,

los dos procesadores realizan la comparación. En ambos casos el coste de comunicaciones es el mismo, pero no el de cómputo. En este caso, igual que en el anterior, todos los paradigmas son posibles, pero OpenMP podría ser el más adecuado para las comparaciones y MPI, para comunicaciones múltiples. Hay diferentes algoritmos de ordenación que implementan estas estructuras y que veremos en seudo código a continuación.

6.1. Algoritmo Merge Bitonic

Es un algoritmo muy simple (Batcher 1968) que considera una secuencia bitónica a una serie de elementos <a0, a1, a2,…, an-1> si existe un índice y si <a0, a1,…, ai> es creciente y monótono y <ai+1,…,an-1> es decreciente y monótono, o que existe un desplazamiento cíclico de los índices si se cumple lo anterior.

Figura 23



Si consideramos como propiedad (de toda secuencia bitónica de *n* elementos) que la operación comparar-intercambiar con los elementos ai y a(i+n/2), obtendremos dos secuencias bitónicas con los números de una secuencia menores que los de la otra. Por lo cual, aplicando recursivamente el comparar-intercambiar a las subsecuencias, llegaremos a listas bitónicas de tamaño uno y así la lista inicial de *n* elementos estará ordenada. Consideremos:

Secuencia original: 3 5 8 9 10 12 14 20 95 90 60 40 35 23 18 0
1.a Comparar-intercambiar: 3 5 8 9 10 12 14 0 | 95 90 60 40 35 23 18 20
2.a Comparar-intercambiar: 3 5 8 0 |10 12 14 9 | 35 23 18 20 |95 90 60 40
3.a Comparar-intercambiar: 3 0 | 8 5 |10 9 | 14 12 | 18 20 |35 23 |60 40| 95 90
3.a Comparar-intercambiar: 0 | 3 | 5 | 8 | 9 | 10 | 12 | 14 | 18 | 20 | 23 | 35 | 40 | 60 | 90 |95

Como se demuestra, el algoritmo funciona pero presenta un problema: necesitamos que la secuencia de entrada cumpla unas propiedades que normalmente no va a cumplir. Por lo tanto, deberemos realizar el paso previo de obtener secuencias bitónicas de un número mayor de elementos cada vez, ya que una secuencia de dos elementos siempre es bitónica. Es decir, cualquier secuencia de *n* elementos se puede ver como la concatenación de un conjunto de secuencias bitónicas de 2 elementos. Deberemos tener en cuenta que al

concatenar las partes ascendentes y descendentes de dos secuencias bitónicas se obtendrá otra secuencia bitónica.

Si consideramos el caso en el que en cada procesador sólo hay un elemento de la secuencia a ordenar, se puede organizar muy fácilmente en arquitecturas regulares como el hipercubo o una malla. Por ejemplo, en un hipercubo los procesadores que difieren en un bit son vecinos, por lo cual en el iésimo-paso, los procesadores que difieran en el (d-i+1) bit realizarán una operación *compare-exchange*. En la malla hay una conectividad menor que el hipercubo, pero buscaremos que la mayoría de las operaciones de comparar-intercambio (pero no todas) se realicen entre vecinos físicos. Por ejemplo, el algoritmo en seudocódigo para el hipercubo será:

```
// label: identificador del procesador = secuencia binaria de "d" bits.
// d: número de dimensiones = número de vecinos de cada procesador.
// comp_exchange_max(j): compara el elemento local con el elemento del procesador más
// cercano a través de la j-ésima dimensión y se queda con el mayor.
    Bitonic_sort(label, d) {
        Para i:=0 hasta (d-1) {
            Para j:=i hasta 0 con decremento (-1) {
                if (nésimo((i+1),label) <> (nésimo(j,label)))
                    comp_exchange_max(j);
                else
                    comp_exchange_min(j);
                }
            }
```

En este caso, la complejidad será $O(\log^2 n)$. Como conclusión, para la selección de un algoritmo deberemos no sólo tener en cuenta el paradigma de programación, sino además analizar la complejidad, las prestaciones y el *speedup* posible de cuestiones relacionadas con la arquitectura subyacente. En este caso, es evidente que el paso de mensajes, rpc u objetos remotos podrían ser los paradigmas más aconsejables (desde más bajo nivel a más abstracción del problema).

6.2. Algoritmo de la burbuja

Éste es otro tipo de algoritmo inherentemente secuencial con (n-1) iteraciones en el peor caso y con un complejidad $O(n^2)$. Para nuestros objetivos, analizaremos una versión transposición par-impar. Esta versión consta de *n* fases, y en cada una de ellas se hacen O(n) comparaciones, lo cual implica $O(n^2)$. El algoritmo distingue entre fases impares y fases pares, donde enla fase impar se comparan (a1, a2), (a3, a4),..., (an-1,an) y en la fase par se comparan (a2, a3), (a4, a5),..., (an-2, an-1). Si se tiene un elemento por procesador, en cada fase se hace una comparación con el vecino inmediato, lo cual significa O(1), y el tiempo de ejecución distribuida/paralela será O(n) que, considerando el conjunto $Tp = O(n^2)$, no será óptima en cuanto al coste. Si tenemos más de un elemento por procesador, existirán p bloques de (n / p) elementos que se ordenan de forma local. A continuación, se hacen *p* fases, donde la mitad sea par

y la mitad impar. En cada fase se hacen O(n / p) comparaciones y O(n / p) comunicaciones, por lo cual una medida del tiempo invertido será:

```
Tp = orden local + comparación + comunicación = O( (n / p) * log (n / p) ) + O(n) + O(n).
```

Lo cual será una operación óptima en coste, pero poco escalable. En seudocódigo será:

```
Impar-Par(n, id) {
    Para i:=1 hasta n {
        if impar(i) {
            if impar(id) compare_exchange_min(id+1)
            else compare_exchange_max(id-1)}
        else {
            if par(id) compare_exchange_min(id+1);
            else compare_exchange_max(id-1) }
    }
}
```

Secuencia inicial: 3 2 3 8 5 6 4 1
1.a fase par: 2 | 3 3 | 8 5| 6 1 | 4
2.a fase impar: 2 3 | 3 5| 8 1 | 6 4
3.a fase par: 2 | 3 3 | 5 1 | 8 4 | 6
4.a fase impar: 2 3 | 3 1 | 5 4 | 8 6
5.a fase par: 2 | 3 1 | 3 4 | 5 6 | 8
6.a fase impar: 2 1 | 3 3 | 4 5 | 6 8
7.a fase par: 1 | 2 3 | 3 4 | 5 6 | 8
8.a fase impar: 1 2 | 3 3 | 4 5 | 6 8

Si bien este algoritmo funciona adecuadamente para un elemento, la transposición par-impar mueve los elementos paso a paso y esto implica demasiados movimientos y sería deseable poder recorrer distancias más largas. Existe una variación de este algoritmo llamado ShellSort que soluciona estos problemas y que puede ser aplicado tanto a arquitecturas regulares como a las distribuidas. El paradigma más adecuado debido a estos movimientos de datos podría ser OpenMP, sin embargo MPI no es descartable, dado que permite operaciones distribuidas sincrónicas.

6.3. Algoritmo Radix

Este algoritmo se basa en la representación en cifras de los números a ordenar, considerando que sean *b* cifras el número de dicha representación. El algoritmo Radix analiza por pasos *r* cifras de representación de un elemento. El algoritmo necesitará b/r iteraciones y durante la iteración *i*, ordenará los elementos en base a esas i cifras más significativos. Si dos bloques tienen la misma i cifra más significativos, mantiene el orden existente entre ellos (se dice que el algoritmo es estable). El tiempo estimado será:

```
Tp = (b / r) * 2 r * (O(log n) + O(n))
```

Inicial: 179 208 306 093 859 984 055 009 271 033

1.º: 1: 271 3: 093 033 4: 984 5: 055 6: 306 8: 208 9: 179 859 009
Resultado: 271 093 033 984 055 306 208 179 859 009
2.º: 0: 306 208 009 3: 033 5: 055 859 7: 271 179 8: 984 9: 093
Resultado: 306 208 009 033 055 859 271 179 984 093
3.º: 0: 009 033 055 093 1: 179 2: 208 271 3: 306 8: 859 9: 984
Resultado final: 009 033 055 093 179 208 271 306 859 984
En seudo código sería:

```
for (i=0; i<n; i++){ // Para cada número
    x=0;
    for (j=0; j<n; j++) if (a[i] > a[j]) x++ //cuento los menores que él
    b[x] = a[i] // copiamos el número en el lugar correcto
    }
```

En este caso, comparar un número con n-1 requiere al menos n-1 pasos, por lo cual para *n* números necesitará n*(n-1) pasos y la complejidad será $O(n^2)$.

Con *n* procesadores podríamos hacer que cada procesador sea el encargado de encontrar la posición final de cada número. Con todos los procesadores trabajando en paralelo, la complejidad sería de O(n). Esta complejidad es inferior a cualquier algoritmo de ordenación secuencial y puede ser mejorada (por ejemplo con el Rank Sort O(log n) con n^2 procesadores).

```
forall (i=0; i<n; i++){ // Para cada número en paralelo
    x=0;
    for (j=0; j<n; j++) if (a[i] > a[j]) x++ //contamos los menores que él
    b[x] = a[i] // copiamos el número en el lugar correcto
    }
```

Es evidente que el paradigma más adecuado es *multithreading* (OpenMP), pero existen implementaciones muy eficientes realizadas en MPI.

6.4. Conclusiones

Como se ha visto en los algoritmos anteriores, el tiempo de cómputo será proporcional a la complejidad y el programador de la aplicación deberá evaluar con detenimiento estas cuestiones para ver la implementación más eficiente desde el punto de vista algorítmico y seleccionar más adelante el paradigma de programación que mejor se adapte a la solución adecuada. La complejidad permite, antes de programar el algoritmo, "tener idea" de cómo funcionará (tiempo estimado de retorno y de respuesta) y lo eficiente que será.

Si no se encuentra el algoritmo específico, se pueden considerar diferentes estructuras para organizar el código paralelo distribuido que presentan determinadas garantías (y se utilizan frecuentemente) y de las cuales se puede extrapolar parámetros de rendimiento y/o eficiencia. Entre ellas podemos enumerar:

1) *Master-worker*: a veces se la llama también *granja de tareas* (*task-farm*), que se adaptan mejor a un tipo de paradigma de programación que a otro y, como

ya se ha visto en *Multithreading*, en un *thread*, el maestro envía trabajo a los trabajadores que los resuelven y devuelven el resultado al maestro.

2) SPMD (*single program multiple data*): al inicio de la aplicación se crea un conjunto de tareas con el mismo código, pero cada una de ellas tiene diferentes datos. El problema de este tipo de organización puede ser la distribución de los datos.

3) *Pipelining*: se realiza una descomposición funcional de las tareas a realizar y se ejecutan en etapas, y una tarea debe terminar antes de comenzar la siguiente. Cada etapa ejecuta la misma tarea en forma concurrente con las restantes etapas/procesos.

4) *Divide & conquer*: se divide el problema en subconjuntos y se solucionas éstos para, finalmente, juntar todos los resultados y obtener el final.

Una vez desarrollado el algoritmo en una u otra tecnología (y haciendo la inversión en diseño, desarrollo y depuración), se puede ver con más detalle la adecuación de la solución desarrollada analizando los siguientes índices:

1) Ratio de ejecución: mide la salida de resultados en función del tiempo (por ejemplo, MIPS o FLOPS para una determinada máquina distribuida/paralela).

2) *Speedup*: utilizando *p* procesadores es la relación:

Speepup (p) = TiempoSerie/Tiempo Paralelo(p)

3) Eficiencia: proporciona una idea del *speedup* respecto a *speedup* lineal y se basa en la idea de que sólo con un sistema paralelo ideal de *p* procesadores se obtendrá un *speedup* lineal igual a *p*.

Eficiencia(p) = speedup(p)/p

4) Redundancia: es el cociente entre el número de operaciones realizadas en un sistema paralelo de *p* procesadores y el número de operaciones necesarias para realizar la misma tarea sobre un sistema monoprocesador.

5) Coste: tiempo de ejecución paralela por el número de procesadores (en un sistema monoprocesador es el tiempo de ejecución).

6) Escalabilidad: relaciona la dependencia de las prestaciones con el número de procesadores y el grado de paralelismo del problema.

Glosario

***application program interface* (API)** *f* Colección de llamadas a subrutinas que permiten a las aplicaciones usar un sistema software.

***atomicity, consistency, isolation and durability* (ACID)** *m* Principales requisitos para un correcto procesamiento de transacciones.

asíncrono *adj* No se garantiza coincidencia en el tiempo de reloj. Por ejemplo, en una comunicación asincrona, el emisor y el receptor pueden, o no, estar involucrados en la operación en el mismo instante de tiempo.

broker *m* Programa (componente o capa software) que acepta peticiones de una capa software o un componente y las trasalada en una forma que sea comprendida por otras capas o componentes.

***common object request broker architecture* (CORBA)** *f* Arquitectura que permite que los objetos puedan comunicarse entre sí, con independencia del lenguaje de programación usado en el que fueron escritos, o del sistema operativo sobre el cual se ejecutan.

componente *m* Pieza de software que puede ser usada como bloque de construcción de sistemas mayores.

computador distribuido *m* Un computador construido por varios menores (y potencialmente independientes), como por ejemplo una red de estaciones de trabajo o computadores de sobremesa. Es destacable la buena relación de coste y eficiencia que se consigue y la flexibilidad del sistema para el crecimiento y actualización. Normalmente, son de tipo heterogéneo.

invocación de operaciones remotas (RPC) *f* Protocolo que permite a un programa ejecutar otro programa (o parte de él), en otra máquina enviando los datos por la red. La situación remota puede ser explicita al programador o transparente. El protocolo se encarga de hacer llegar los datos, invocar las operaciones y devolver los resultados a quien invocó las operaciones.

***marshalling*/serialización** *m* Proceso de recoger datos y transformarlos en un formato intermedio estandar con objeto de transmitirlo por red a otras máquinas. Tanto emisor como receptor necesitarán procesos de transformación de sus datos originales a/desde el formato intermedio, para obtener los datos equivalentes en su representación.

memoria compartida *f* Memoria que le aparece al usuario conteniendo un único espacio de direcciones y a la que puede accederse por medio de cualquier proceso.

memoria distribuida *f* Memoria que está fisicamente distribuida entre varios módulos. Una arquitectura distribuida de memoria puede aparecer a los usuarios como una única memoria compartida con un espacio de direcciones único, o puede aparecer como memoria disjunta con varios espacios de direcciones.

***message oriented middleware* (MOM)** *m* Categoría de comunicación entre aplicaciones que, en general, se basa en el paso de mensajes de forma asíncrona. La mayoría de *middleware* basados en MOM depende de una cola de mensajes del sistema, aunque algunas implementaciones dependen de sistemas de emisión (*broadcast*), o de sistemas de multidifusión (*multicast*).

***message passing interface* (MPI)** *f* Interfaz de paso de mensajes como protocolo de comunicación entre computadores (o entre procesos en el mismo computador), que define un estandard (de la organización MPI Forum) de comunicación entre procesadores que ejecutan una aplicación en un sistema de memoria distribuida.

middleware *m* Capa de software distribuida que se sitúa sobre el sistema operativo (de red) y antes de la capa de aplicación, abstrayendo la heterogeneidad del entorno.

MIMD *f* Multiples instrucciones, multiples datos. Categoría de la taxonomía de Flynn, en la cual múltiples flujos de instrucciones son concurrentemente aplicados a múltiples conjuntos de datos. Una arquitectura MIMD es aquélla donde multiples procesos heterogéneos pueden ejecutarse a diferentes ratios.

multicomputador *m* Ordenador cuyos procesadores pueden ejecutar flujos de instrucción separados, tienen sus propias memorias privadas, y no pueden acceder directamente a la memoria de unos a otros. La mayoría de multicomputadores son máquinas de memoria disjunta construidas juntando nodos vía enlaces de red de comunicaciones.

multiprocesador *m* Ordenador cuyos procesadores pueden ejecutar flujos separados de instrucciones, pero disponen de un único espacio de direcciones. La mayoría son máquinas de memoria compartida, construidas conectando varios procesadores con memorias mediante una arquitectura de bus, malla o *switch*.

multitarea *f* Múltiples procesos ejecutándose en un único procesador. Normalmente, es una característica del sistema operativo, que permite repartir el tiempo de ejecución del procesador entre las ejecuciones de las tareas individuales, realizando cambios de contexto en cada intercambio de proceso en ejecución.

multithreading *en* Véase **thread**.

***non uniform memory access* (NUMA)** *m* Sistemas donde las memorias no disponen de tiempo uniforme para las operaciones de lectura o escritura. En sistemas NUMA, la memoria se construye normalmente de forma jerárquica; en sistemas con multiprocesador, algunas porciones de la memoria pueden leerse más rápidamente que otras.
Véase UMA

***open multi processing* (OpenMP)** *f* Interfaz de programación (API) con soporte multiplataforma para la programación en C/C++ y Fortran de procesos, que utiliza memoria compartida sobre plataformas UNIX y Windows.

paradigma de programación *m* En la generación de código fuente representa la realización bajo un enfoque particular o bien cuestiones de índole filosófica sobre la construcción del software.

ratio comunicación-cómputo *f* Ratio del número de cálculos desarrollados por un proceso frente al tamaño total de los mensajes que éste envia. La ratio tambien se suele calcular como la usada anteriormente para computación frente a la usada para comunicación.

servicio web *m* Colección de protocolos y estándares que sirven para intercambiar datos entre aplicaciones. Distintas aplicaciones de software desarrolladas en lenguajes de programación diferentes, y ejecutadas sobre cualquier plataforma, que pueden utilizar los servicios web para intercambiar datos en redes de ordenadores como Internet. La interoperabilidad se consigue mediante la adopción de estándares abiertos. Las organizaciones OASIS y W3C son los comités responsables de la arquitectura y reglamentación de los servicios web.
en web service

SIMD *m* Simple instrucción de multiples datos, categoría de la taxonomía de Flynn, donde un flujo de instrucciones es concurrentemente aplicado a múltiples conjuntos de datos. Una arquitectura SIMD es aquélla en la que procesos homogéneos sincrónicamente ejecutan las mismas instrucciones en sus propios datos.

SPMD *m* Programa simple, múltiples datos; una categoría añadida a la taxonomía de Flynn, en la cual se describen programas compuestos por varias instancias de un tipo único de proceso, cada uno ejecutando el mismo código de forma independiente. Puede verse como una extensión de SIMD o una restricción de MIMD.

SOA *f* Aproximación arquitectural para construir aplicaciones que implementan procesos de negocios o servicios, usando un conjunto de componentes de caja negra débilmente acoplados y organizados para ofrecer un nivel de servicio definido.

SOAP *m* Protocolo simple de acceso a objetos, que permite, con independencia de la máquina y de las particularidades de la red, utilizar un vocabulario basado en XML, común para codificar datos y operaciones.

taxonomía de Flynn *f* Esquema de clasificación arquitectural con dos ejes: el número de flujos de instrucciones ejecutándose concurrentemente, y el número de conjuntos de datos sobre los que se aplican esas instrucciones.

thread *m* Secuencia de ejecución (hilo) de un programa, es decir, las diferentes partes o rutinas de un programa que se ejecutan concurrentemente en un único procesador. Generalmente, los *threads* están contenidos en un proceso, y diferentes hilos de un mismo proceso pueden compartir algunos recursos, mientras que diferentes procesos, no. La ejecución de múltiples hilos (*multithreading*) en paralelo necesita soporte del sistema operativo, y el soporte físico de varios procesadores, o en su defecto, el soporte multiprogramado del sistema operativo en un único procesador para el soporte concurrente de los hilos.

UMA (*uniform memory access*) *m* Permite a cualquier elemento de memoria ser leído o escrito en el mismo tiempo de forma constante.

web service Véase **servicio web**.

Bibliografía

Sinha, P. (1997). *Distributed Operating Systems: Concepts & Design*. IEEE Press.

Tanembaum, A.; Steen, M. (2007). *Distributed Systems: Principles and Paradigms*, 2/E. Prentice Hall.

Coulouris, G.; Dollimore, J.; Kindberg, T. (2005). *Distributed Systems: Concepts & Design* (4.a ed.). Addison-Wesley. [traducció al castellà: *Sistemas distribuidos: conceptos y diseño* (2001, 3.a ed.). Pearson.]

Birman, K. P. (2005). *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer *Middleware for Communications*. Qusay Mahmoud, Wiley 2004.

Andrews, G. R. (1999). *Foundations of Multithreaded, Parallel, and Distributed*. Addison-Wesley.

Wilson, G. V. (1993). *A Glossary of Parallel Computing Terminology*. IEEE Parallel & Distributed Technology.

Quinn, M. J. (2003). *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill.

Foster, I. (1995). *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley.

Goetz, B. (2006). *Java Concurrency in Practice*. Addison-Wesley.

Lea, D. (2000). *Concurrent Programming in Java*. Addison-Wesley.

Pacheco, P. (2000). *Parallel Programming With MPI*. Morgan Kaufmann.

Stevens, W. R. (2005). *Advanced Programming in the UNIX Environment* (2.a ed.). Addison-Wesley Professional.

Sridharan, P. (1997). *Advanced Java Networking*. Prentice Hall.

Oberg, R. (2001). *Mastering RMI: Developing Enterprise Applications in Java and EJB*. John Wiley & Sons.

Burke, B. (2006). *Enterprise JavaBeans 3.0*. O'Reilly Media.

Cornell, G.; Horsmann, C. (1996). *Core Java*. Prentice Hall.

St. Laurent, S. (2001). *Programming Web Services with XML-RPC*. Reilly Media.

Cerami, E. (2002). *Web Services Essentials (O'Reilly XML)*. O'Reilly Media.

Pew, J. (1996). *Instant Java*. Prentice Hall.

Englander, R. (2002). *Java and SOAP*. O'Reilly Media.

Barclay, K.; Gordon, B. (1994). *C++, Problem Solving & Programming*. Prentice Hall.

Páginas web

Tutoriales de la tecnología Java: http://java.sun.com/docs/books/tutorial/

Java Beans: http://java.sun.com/products/javabeans/

RMI: http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

W3C, Web Services Architecture: http://www.w3.org/TR/ws-arch/

JAX-WS: https://jax-ws.dev.java.net/

MPI: http://www-unix.mcs.anl.gov/mpi/

MPICH2: http://www-unix.mcs.anl.gov/mpi/mpich2/

DCOM: http://www.microsoft.com/com/default.mspx

Java Developers: http://www.developer.com/java/

IBM WS: http://www-128.ibm.com/developerworks/webservices/standards/