Anales PANEL'81/12 JAIIO
Sociedad Argentina de informática
e Investigación Operativa. Buenos Aires, 1981

REFLEXIONES SOBRE PROGRAMACION CONCURRENTE Y PROGRAMACION DISTRIBUIDA

Roberto Pardo Silva

Instituto SER de Investigación
Apartado Aéreo 1978, Bogotá - Colombia.

OBJETIVO

Es bien sabido que el costo del desarrollo de software (1) continúa aumentando mientras que el de su contraparte, hardware, continúa disminuyendo. Así, la dependencia en tecnología de computadores en muchos paises se ha convertido y se convertirá aún más en una dependencia de software. Por esta razón es imperioso que en paises como los latinoamericanos se comience a dominar la complejidad del desarrollo de software, no solo cuando este se implante en arquitecturas convencionales como las centralizadas, sino cuando se desee implantar en arquitecturas sofisticadas como las distribuidas.

Este artículo se concentra en discutir conceptos fundamentales usados en lenguajes de programación concurrente e ilustrar problemas claves que tendrán que atacarse al diseñar lenguajes de programación distribuida.

TERMINOS CLAVES: Procesamiento Distribuido, Programación Concurrente, Programación Distribuida.

(1) Se usarán los términos "software" y "hardware" del inglés por no existir una traducción ampliamente aceptada de tales términos

I. INTRODUCCION

La tendencia a desarrollar sistemas de información descentralizados, es decir sistemas donde los recursos están dispersos en múltiples máquinas interconectadas entre sí, se ha ido acentuando en los últimos años. Varios factores contribuyen a este fenómeno:

1) En todas las organizaciones relativamente grandes, las decisiones basadas en información las toman diferentes personas. En muchos casos los generadores y usuarios de información están distantes. En vez de implantar un sistema de información en una sola máquina, el cual muchas veces tiende a alterar la estructura natural de la organización, es deseable implantar un sistema de información descentralizado que se adapte a la organización y permita a los usuarios tener control más directo de su información(v. g., el grupo de estadística recolecta, procesa, y analiza la información estadística sin depender de "sistemas"), delinear las responsabilidades más claramente (v. g., evita conflictos como aquellos cuando el grupo de estadística culpa al grupo de sistemas y viceversa si algo no funciona), suprimir cuellos de botellas y conflictos de prioridades (v. g., el grupo de sistemas y la máquina misma se recarga de trabajo y no es claro a qué parte de la compañía se debe dar prioridad ya que cada parte considera independientemente ser muy importante), planear más fácilmente aumentos de capacidad en el sistema(v. g., en una sóla máquina los cambios pueden afectar muchas aplicaciones mien tras que si existen varias máquinas cada una corriendo pocas aplicaciones se evitan tales efectos), etc.

2) El costo del hardware ha disminuído tremendamente. El costo de procesadores, memorias, etc, ha disminuído órdenes de magnitud hasta el punto que varias máquinas pueden ser similares en costo al de una sóla con poder computacional equivalente al agregado de varias máquinas. Además si se cuantifican las ventajas en claridad (una máquina "grande" es mucho más confusa y su software menos confiable que en las pequeñas), confiabilidad (la disponibilidad de un sistema de varias máquinas es típicamente mayor a la de un sistema con una sóla máquina), facilidad de expansión (un sistema de varias máquinas es forzosamente modular y la inversión inicial puede ser mínima), etc, el costo de un sistema de varias máquinas puede ser aún más competitivo con el de una sóla equivalente.

3) La tecnología de interconexión de máquinas, es decir, redes de computadores, ha avanzado a pasos gigantezcos en la última década. La red ARPA (DAVI79), es la pionera en la tecnología de interconexión de múltiples máquinas: se implementa el concepto de conmutación de paquetes, se desarrollan jerarquías bien definidas de protocolos, se experimentan en algoritmos de enrutamientos novedodos, etc. Los adelantos gene-

rados en gran parte por el éxito de la red ARPA son numerosos: la mayoría de los paises desarrollados están en proceso si no lo han hecho ya, de ofrecer redes públicas de datos con conmutación de paquetes; casi todos los fabricantes ofrecen arquitecturas de redes (v. g., SNA de IBM, DECNET de Digital , XODIAC de Data General, BNA de Borroughs, etc), existen experimentos exitosos del uso de redes con transmisión eficiente vía satélite y redes moviles que transmiten por ondas de radio; varias redes se han inteconectado entre sí; las redes locales han dejado su fase experimental para convertirse en productos (v. g., Z-NET de ZILOG, ETHERNET de Xerox/DEC/INTEL, RINGNET de Prime, etc). En síntesis , en la década de los 80 las arquitecturas de múltiples máquinas interconectadas entre sí serán las arquitecturas en las cuales se tendrá que implantar muchos sistemas de información.

El aspecto central de este artículo no es justificar la existencia de este tipo de arquitecturas de múltiples computadores. El punto de interés sin embargo, es tratar de reflexionar sobre las posibilidades de desarrollar buen software en tales arquitecturas. Específicamente se tratará de contestar en este artículo interrogantes como, qué conceptos de lenguajes tradicionales para arquitecturas de una sola máquina son válidos en arquitecturas de varias máquinas? Cuáles deben ser las características de un lenguaje de programación que permita expresar algoritmos para implantarse en una arquitectura de varias máquinas, es decir, algoritmos distribuidos?

Se comienza en la siguiente sección con una taxonomía de modelos de programación y consecuentemente de lenguajes de programación. En la sección 3, se menciona brevemente el modelo de programación del cual todos estamos familiarizados, es decir programación secuencial. En la sección 4 se discute en detalle los conceptos fundamentales usados en programación concurrente para contrastarlos en la siguiente sección la 5, con los problemas encontrados en programación distribuida. Finalmente en la última sección se enumeran una serie de conclusiones y se sugieren tópicos de investigación.

2. MODELOS DE PROGRAMACION

Para entender el fenómeno de programación en arquitecturas de varias máquinas y relacionarlo con la programación convencional, examinaremos brevemente en qué consiste la actividad de programar. Dado un problema se desea hallar una solución algorítmica. Típicamente el problema inicial se divide en subproblemas de menos complejidad y sucesivamente refinamos soluciones a estos problemas hasta producir un algoritmo detallado. Generalmente el final de tal refinamiento consiste en representar el algoritmo detallado en algún lenguaje de programación , es decir se produce un programa. Un programa , por lo tanto , es una de muchas representaciones a la solución de un problema.

Por otro lado, todo lenguaje de programación supone la existencia de

un ambiente de ejecución el cual es simplemente el esquema usado para interpretar las instrucciones del programa durante su ejecución. El ambiente de ejecución es importante para quien diseña e implanta el lenguaje, pero también repercute, como lo podrá apreciar el lector más adelante, en el tipo de algoritmos (y consecuentemente de problemas) que se pueden representar (resolver) .

Aclaremos lo anterior usando dos ejemplos bastante familiares. Sea el problema "buscar si un número está en una tabla ordenada" y sea el algoritmo "búsqueda binaria" . Primero pensemos en la representación de tal algoritmo en FORTRAN. El ambiente de ejecución en este caso consiste en elementos tales com el procesador de instrucciones de la máquina para la cual se compiló el programa, un espacio fijo de memoria, etc. Si por el contrario la representación del mismo algoritmo, se hace en ALGOL-60 para resolver el mismo problema, el ambiente de ejecución consiste en elementos como el procesador de instrucciones de la máquina para la cual se compiló el programa, un espacio variable de memoria, un "stack" , etc.

Aunque los dos ejemplos anteriores difieren en características importantes como la capacidad de asignación dinámica de memoria, para efectos de nuestra discusión son idénticos en cuanto a sus componentes básicos de hardware. Es decir, ambos ambientes de ejecución suponen la existencia del par procesador/memoria como los elementos primordiales para una realización física en hardware que sirva de soporte a programas en tales lenguajes.

Aclarado nuestro concepto de componentes básicos del hardware de un ambiente de ejecución, vamos a proponer en este artículo una caracterización de cuatro clases de modelos de programación, cada uno de los cuales definiendo a su vez un conjunto de lenguajes de programación (ver tabla 1).

3. PROGRAMACION SECUENCIAL

Sin duda alguna este es el modo de programación más frecuentemente usado. Se caracteriza por tener un ambiente de ejecución donde se ejecuta una instrucción a la vez; los algoritmos son independientes del tiempo; y el resultado siempre depende solo de la entrada y el programa (o sea un progrma P con unos datos de entrada E siempre producen el mismo resultado R). Los lenguajes típicos usados en programación secuencial son FORTRAN, COBOL, ALGOL-60 SNOBOL PL/I (sin subtasks), PASCAL, etc.

Muchos tipos de soluciones a problemas, sin embargo, no se pueden representar (o realizar) con este tipo de herramientas, Consideremos por ejemplo el problema " encontrar el máximo elemento de una lista de números ". Una solución del tipo "hállese simultáneamente el máximo elemento de cada mitad y luego escójase el máximo entre ellos" no se puede realizar en un lenguaje secuencial (1).

(1) Una inquietud que se deja al lector es, existen problemas no-secuenciales? o la no-secuencialidad es un artificio exclusivo de la solución a problemas?

Para	el componente básico de hardware de su ambiente de ejecución es:
PROGRAMACION SECUENCIAL	(PROC;MEMORIA)(1)
PROGRAMACION CONCURRENTE	(MULTIPLES (PROC);MEMORIA)
PROGRAMACION EN TIEMPO REAL	((MULTIPLES(PROC);MEMORIA)+RELOJ)
PROGRAMACION DISTRIBUIDA	((MULTIPLES ((PROC;MEMORIA)+RELOJ))+ SISTEMA DE COMUNICACION

TABLA 1 Modelos de Programación

(1) Un lenguaje aplicativo como LISP-puro no usaría memoria, solo un procesador en esta taxonomía.

---

4. PROGRAMACION CONCURRENTE

La programación concurrente se usa típicamente para implantar sistemas operacionales, base de datos, protocolos, etc. Además de este tipo de "programas del sistema" también se pueden usar en aplicaciones convencionales tipo "cascada" para aumentar su eficiencia. Por ejemplo consideremos una aplicación (v.g., nómina, contabilidad, control de inventarios, etc.) que se pueda abstraer en tres pasos en cascada: 1) entrada, donde se lee una transacción y se verifica su integridad, 2) procesamiento, donde se hacen cálculos y se actualizan uno o varios archivos, y 3) salida, donde se escribe alguna información del resultado de la transacción.
Si en vez de empezar a correr cada transacción únicamente al final de los pasos se tienen 3 transacciones simultáneamente cada una en un paso diferente, es posible disminuír el tiempo total de ejecución del conjunto de transacciones (v.g., si el tiempo promedio de ejecución de los pasos es comparable, la reducción total puede ser del orden de 1/3).

Este modo de programación se caracteriza por tener un ambiente de ejecución donde potencialmente se pueden ejecutar varias instrucciones a la vez; los algoritmos como tales son independientes del tiempo y el resultado, además de depender de las entradas y el pro-

grama, puede depender de otros factores (v.g., el modo de despacho de la unidades de concurrencia). Los lenguajes concurrentes típicos son PL/I (con subtasks) , ALGOL-68, CONCURRENT PASCAL, MODULA, MESA, ADA, etc.

Existen tres problemas claves que debe resolver todo lenguaje de programación concurrente:

1) El problema de manejo de la unidad de concurrencia, es decir, debe existir un mecanismos sintáctico con su semántica asociada para definir pedazos de código que puedan ejecutar concurrrentemente y mecanismos para activar y desactivar tales unidades de concurrencia.

2) El problema de sincronización, es decir, debe existir un mecanismo sintáctico con su semántica asociada para controlar el orden de ejecución de ciertos eventos y

3) El problema de comunicación, es decir, debe existir un mecanismo sintáctico con su semántica asociada para pasar información entre las unidades de concurrencia.

A continuación exploraremos el problema de sincronización por ser el más importante.

4.1 ALGUNAS SOLUCIONES AL PROBLEMA DE SINCRONIZACION

En principio existen muchos tipos de orden de eventos para controlar en un programa concurrente. Los ejemplos clásicos son ' exclusión mutua' , ' lectores y escritores con diferentes prioridades' , ' productor/consumidor' , etc. En general muchos de estos tipos de eventos suceden cuando las unidades de concurrencia comparten una estructura de datos (v.g., un archivo , un buffer). La idea clave es que un lenguaje de programación concurrente debe proveer buenas herramientas de sincronización (i.e., un modelo de sincronización), donde "buena" quiere decir una combinación de factores como generalidad , facilidad de implantación, naturalidad en su uso (v.g., de alto nivel , estructura , etc.), modularidad, y que se preste a verificaciones formales.

Examinemos entonces el papel que juegan varios modelos de sincronización en lenguajes de programación concurrente. (Las descripciones en detalle se pueden encontrar en diferentes artículos).

MODELOS DE BAJO NIVEL

Sin duda alguna el modelo más clásico es el de P/V originado por E.W. Dijkstra (DIJK68). Básicamente P y V son operaciones en objetos llamados semáforos que permiten a las unidades de concurrencia (procesos) indicar la ocurrencia de eventos . Los semáforos son objetos compartidos por las unidades de concurrencia (observación muy importante cuando se trata de extrapolar la idea de semáforos a un sistema distribuido) que tienen un valor entero y una sola cola asociada . Dado un semáforo s las operaciones definen las siguiente lógica:

```
P(s): if  s o then  s  s-1
                else  poner_en_cola
                      bloquee

V(s): s - s+1;
          active_proceso_si_existe_cola_de_espera
```

Este modelo como tal no está asociado a un lenguaje. Sin embargo, si existen primitivas en lenguajes que realizan funciones similares. Por ejemplo, los macros ENQ/DEQ sobre nombres de recursos en assembler IBM 360-370, los "statements" WAIT/COMPLETION sobre variables 'eventos' en PL/I, etc . Todas estas estructuras son equivalentes y por lo tanto poseen las mismas ventajas y desventajas .

El modelo P/V fué una gran contribución para entender y solucionar problemas de sincronización en sistemas . Sin embargo, su uso como herramienta de programación (o sea, que P y V sean las primitivas que directamente 'vea' el programador) es altamente cuestionable. Específicamente estas primitivas son de muy bajo nivel (a la assembler) y las soluciones tienden a ser obscuras (v.g., ver alguna solución de 'lectores' y escritores' con prioridad que usen semáforos), a ser poco naturales (v.g., nótese en la solución del buffer que escoger el tipo de semáforos no es trivial), a dejar esparcida la sincronización en las unidades de concurrencia en vez de asociarla en el recurso compartido que requiere la sincronización, etc.

Otro modelo muy similar, o sea con el mismo poder, ventajas y desventajas, es el de SEND/RECEIVE (BRIN70)

MODELOS DE ALTO NIVEL

El avance más fundamental en el tipo de herramientas para describir soluciones a problemas de sincronización lo constituyó la introducción de monitores (HOAR74).

Desde el punto de vista de lenguajes, un monitor es un tipo de datos con sus operaciones (procedimientos) asociadas, que normalmente abstrae el recurso compartido. Las operaciones del monitor son mutuamente excluyentes (algo que implanta el ambiente de ejecución y que es transparente para el programador).

La sincronización se realiza usando dos operaciones especiales wait y signal. La primera suspende al proceso (la unidad de concurrencia que como parte de su ambiente ejecuta una operación del monitor) y lo pone al final de una cola (nombrada como parámetro de la operación). La segunda hace que el primer proceso de la cola nombrada se active y que el proceso que invoca la operación quede en una cola de más prioridad que los suspendidos por la primera operación. Existen varias versiones del concepto de monitor. Por ejemplo, en MESA (LAMP80) solo los procedimientos ENTRY de un monitor se excluyen mutuamente y existe una primitiva llamada

broadcast para "despertar" todos los procesos de una cola. Sin embargo, conceptualmente son muy parecidos.

En general se puede afirmar que el monitor es una estructura con poder suficiente para resolver gran variedad de problemas de sincronización. Su inclusión como tipo especial con operaciones definidas por el programador es tal vez lo más significativo. Sin embargo su aspecto más débil es la necesidad de incluír un mecanismo explícito de señalización (los wait's y signal's) dentro del monitor Este mecanismo le da un "sabor a la assembler" donde no se sabe quién va a despertar a quién y le hace perder claridad a la solución. Otros problemas más detallados como el de "deadlocks" al anidar llamada a monitores y el de obligar al programador a fijar ciertas prioridades se pueden encontrar en (BLOO79).

Otro modelo de alto nivel es el de las Expresiones de Camino o "Path Expressions" (CAMP74). Básicamente este mecanismo permite especificar el tipo de sincronización de una abstracción de datos como parte de la definición misma de la abstracción. La idea es controlar el acceso a un recurso (representado en la abstracción) definiendo el orden permisible de las operaciones del recurso.
Así, existe una sintaxis especial para especificar los ordenes permisibles . Por ejemplo, el operador "+" indica selección o exclusión mutua; al operador";" indica secuencia; el operador "}{" indica concurrencia; etc.

Tal vez el aspecto más atractivo de las Expresiones de Camino es su enfoque no-procedimental, es decir, todas las restricciones de sincronización quedan capturadas en una sóla especificación donde el programador solo se preocupa del "qué debe ser la sincronización" y se olvida del "cómo" (en cambio con monitores el programador tenía que internarse en el cómo al usar wait's y signal's). Sin embargo, cuando soluciones requieren cierto tipo de prioridades o son dependientes del estado del recurso, las Expresiones de Camino deben aumentarse para poder resolver los respectivos problemas (BLOO79).

4.2 ALGUNAS LECCIONES APRENDIDAS

Resumiendo brevemente las diferentes soluciones a los problemas claves de programación concurrente se tiene lo siguiente:

a) Sincronización. Inicialmente se identificó el problema de 'exclusión mutua' y aparecieron soluciones usando LOAD/STORE (o lo que es equivalente, sin usar operadores especiales), LOCK/UNLOCK, P/V, SEND/RECEIVE, etc. Aunque estos modelos iniciales no estaban asociados con lenguajes, algunas estructuras similares (y por lo tanto muy primitivas) se infiltraron en lenguajes de alto nivel (v. g., en ALGOL, en PL/I). Luego aparecieron las regiones críticas y los monitores, los cuales representan un gran adelanto por ser buenas herra-

mientas de programación (v.g., en PASCAL CONCURRENTE, en ADA, en MESA). Varios modelos alternos se han propuesto, después del monitor: Expresiones de Camino , serializados el ACCEPT y SELECT de ADA, etc. En general un problema fundamental es el de identificar los tipos de problemas que realmente debe resolver un modelo de sincronización, ya que típicamente cada modelo es 'mejor' que otros ante ciertos tipos de problemas.

b) Unidad de Concurrencia. La construcción que normalmente se usa como unidad de concurrencia es el proceso (subtask en PL/I). Se activa mediante llamadas explícitas , ya sea mediante CALLS especiales (PL/I), INITs en lenguajes donde el proceso es otro tipo de datos (PASCAL CONCURRENTE), o llamados especiales como FORK (MESA).
Típicamente la activación produce estructuras jerárquicas y la unidad de concurrencia acaba su ejecución internamente (v.g., con un RETURN) o externamente por un ABORT de quién lo activó.

c) Comunicación. La implantación de la comunicación entre procesos se hace compartiendo memoria principal. El programador, sin embargo, realiza la comunicación a nivel de lenguaje por medio de estructuras de datos globales, o a través de los mismos mecanismos de sincronización (v.g., el monitor es un área global de comunicación entre procesos, SEND y RECEIVE además de sincronización son obviamente mecanismos de comunicación).

Se puede afirmar entonces que la tendencia primordial es diseñar herramientas estructuradas para resolver cualquiera de estos problemas . La programación concurrente lleva varios años en los cuales se han propuesto muchos tipos de soluciones, cada vez "mejores". Esta observación , desafortunadamente, es inquietante ya que si este es un dominio de problemas relativamente conocidos y sobre los cuales no existe la solución óptima , qué podremos esperar de dominios de problemas bastante desconocidos como los de los ambientes distribuidos?

5. PROGRAMACION DISTRIBUIDA

El advenimiento de sistemas distribuidos ha creado arquitecturas donde al resolver un problema, el algoritmo debe representarse en un lenguaje de tal forma que partes de código ejecutan en diferentes máquinas y a su vez estas partes interactúan entre sí. En los últimos años han aparecido gran cantidad de " algoritmos distribuidos " para resolver problemas de control de concurrencia distribuida (ELLI77, LIND79, THOM79, KANE79), de atomicidad de transacciones distribuidas (LAMP79, REED78), de deadlocks distribuidos, de procesamiento distribuido de 'queries ', etc. Sin embargo , muy poco se ha dicho de las características de un lenguaje donde se pueden representar tales soluciones. En general, la mayoría de los algoritmos se descri-

ben en lenguaje natural o se buscan formalismos gráficos (como en (ELLI77)) que están aún lejos de las descripciones "entendibles" por el sistema (1).

Volviendo a nuestro esquemas de modelos de programación se puede afirmar que la programación distribuida se caracteriza así: el ambiente de ejecución (los ambientes?) ejecutan en paralelo, o sea tiene por lo menos la complejidad de la programación concurrente; el sistema de comunicación introduce retrasos haciendo que los algoritmos dependan del tiempo, o sea tiene adicionalmente la complejidad inherente a la programación en tiempo real; y como si fuera poco muchos programas se hacen para continuar funcionando en el caso de que ciertos pedazos de código sean inaccesibles (debido a fallas de comunicación o de las máquinas mismas). Es fácil deducir que en este ambiente un resultado depende de muchos factores además de la entrada y el programa mismo.

Entre las propuestas más cercanas a lenguajes de programación distribuida están PLITS (FELD79), DISTRIBUTED PROCESSES (BRIN78), extensiones a CLU (LISK79), MOD (COOK79), etc.

Siguiendo el marco de referencia de este artículo sobre modelos de programación se plantean varios problemas claves que deben resolver los lenguajes de programación distribuida:

a) El problema de comunicación remota, es decir, deben existir mecanismos para intercambiar información entre pedazos de código distantes,

b) el problema de sincronización distribuida, es decir, deben existir mecanismos para controlar el orden de eventos que pueden suceder en máquinas distantes,

c) el problema de nombres entre nodos, es decir, deben existir mecanismos para asociar y reconocer nombres que se usen en programas remotos (v.g., nombres globales, nombres de entidades como procedimientos, procesos o módulos que sean 'exportables' a través de nodos) sin atentar contra la autonomía de un nodo,

d) el problema de confiabilidad, es decir, deben existir mecanismos para manipular estructuradamente las fallas de comunicación o de los nodos y el reinicio de éstos, y

e) el problema de tiempo, es decir, deben existir mecanismos para manipular el tiempo, el cual avanza independientemente en cada nodo.

(1) Es obvio que es útil tener formalismos de descripción 'lejanos' al sistema. Sin embargo, adicionalmente se deben tener formalismos de descripción 'cercanos' a la máquina, si nuestro interés es implantar sistemas.

El problema fundamental al resolver estos problemas (obviamente habrá otros) es el de entender qué quiere decir una buena solución , o sea una solución estructurada, flexible, que resuelva los "problemas típicos" , que se pueda implantar eficientemente, etc.

5.1 LOS PROBLEMAS DE UN AMBIENTE DISTRIBUIDO

Todo modelo de programación se basa en una serie de suposiciones con respecto al ambiente de ejecución de sus programas. Un buen entendimiento de estas suposiciones es básico para entender el tipo de problemas de interés . En particular, es interesante anotar cómo ciertos diseños en lenguajes concurrentes se han extrapolado a ambientes distribuidos, llevando implícitamente una serie de suposiciones ' fuertes ' sobre estos últimos ambientes.

Consideremos, por ejemplo, un lenguaje concurrente donde los procesos se comunican a través de mensajes . Aunque se supone que un mensaje siempre llega a su destino no se puede predecir el tiempo de retraso. Bajo este esquema (similar a lo propuesto inicialmente en (FELD79, HOAR79, BRIN78) se puede pensar que existe una gran similitud con un ambiente distribuido (v. g., no hay ambiente global y la comunicación se hace por mensajes que llegan a su destino con un retraso ilimitado pero finito.)

Esta manera de enfocar el problema (respetable por supuesto ya que ignora ciertos problemas para concentrarse en otros también importantes), desafortunadamente olvida la esencia fundamental del software distribuido, es decir, la de implantar sistemas que proveen la ilusión de ser muy confiables lo cual implica que debe atacar el problema de fallas. Dicho de otra forma , se deben diseñar herramientas que no supongan la existencia de sistemas muy confiables ya que precisamente mediante estas herramientas el diseñador desea implantar sistemas confiables. Aclaremos un poco más las características de un ambiente distribuido para entender el tipo, de suposiciones que se deben hacer en un modelo de programación distribuida.

En un ambiente distribuido una computación se modela (al igual que ambientes centralizados concurrentes) como un conjunto de procesos que cooperan. En ambientes centralizados, sin embargo, la comunicación se implanta compartiendo memoria principal. En este esquema la comunicación es casi instantánea , no se pierden mensajes y si existe una falla , casi siempre es tan catastrófica que, se aborta la computación y se comienza de nuevo.

En un ambiente distribuido los procesos son distantes. Dado que los mensajes ' viajan ' por un sistema de comunicación , existe un retraso en la comunicación. Tales retrasos tienen implicaciones en la eficiencia de la computación (v. g., construír un estado global es poco factible ya que, si se para el avance en cada nodo es demorado y se hace poco trabajo, y si no se para el avance, cuando se acaba de construír el estado ya no refleja la situación actual) y en el manejo del tiempo (v. g, cada nodo debe ser capaz de esperar por la ocurren-

cia de eventos hasta cierto límite).

Por otro lado, el sistema de comunicación de un ambiente distribuido aisla fallas, es decir, toda una máquina puede fallar sin que fallen otras. Así mismo, las computaciones se diseñan para seguir funcionando cuando ciertas partes sean inaccesibles y típicamente estas fallas se detectan mediante el manejo del tiempo (v. g., si después de cierto tiempo e intentos un nodo no se puede comunicar con otro, lo declara inoperante).

Estas características definen por lo tanto un modelo de programación distribuida que supone retrasos, existencia de fallas, e incluye la noción del tiempo.

5.2 UN EJEMPLO CONCRETO

Consideremos un ejemplo para poder analizar en concreto algunos problemas en lenguajes de programación distribuida.

El problema de interés se conoce con el nombre de "atomicidad de transacciones distribuidas ". Este se presenta en bases de datos (o inclusive en sistemas de archivos) distribuidos donde se tienen varios archivos , cada uno en un nodo o máquina diferente, y se desea que una transacción (o sea un conjunto bien definido de lectura y/o escrituras a los archivos y que tienen un sentido lógico para el usuario) actualice varios de estos archivos. La propiedad de atomicidad garantiza, que todas o ninguna de las actualizaciones de la transacción eventualmente se efectúa . Por ejemplo, sea la siguiente transacción en una aplicación bancaria que asigna a sucursales de un banco en otras ciudades, un porcentaje de un monto total de dinero para los préstamos locales. Supongamos que cada sucursal tiene sus archivos propios donde está grabada la información del monto local para préstamos.

```
Transacción préstamos (monto_total , %_A, %_B ,%_C )
        if %_A +%_B+%_C≠1 then aborte
        saque monto_total de principal;

        asigne %_A*monto_total a sucursal en ciudad A;
        asigne %_B*monto_total a sucursal en ciudad B;
        asigne %_C*monto_total a sucursal en ciudad C;

Fin transacción
```

En este ejemplo la sintaxis de nuestra transacción no es importante. Los relevante es que existe un conjunto de actualizaciones (saque' s y asigne' s) en nodos diferentes que deben realizarse completamente o no deben realizarse en lo absoluto (e. g., sería grave ' sacar' el monto total de la principal y que sólo ciertas sucursales reciban la ' asignación').

Una solución a este problema es el algoritmo del " compromiso en

dos fases" (LAM79, LIND79). A continuación describimos una versión simplificada de la solución (se sugiere al lector interesado consultar en detalle las referencias anteriores ya que existen gran cantidad de sutilezas asociadas con el algoritmo.)

La transacción se ejecuta en el nodo donde se origina (sea este el nodo coordinador). Las actualizaciones de la transacción sobre datos residentes en varios nodos, no se efectúan directamente sobre los datos sino en algún área extra (v.g., se escribe un ' track ' nuevo del disco que contiene los mismos datos del viejo pero ya actualizados). Con este mecanismo es fácil minimizar el tiempo para hacer efectivas varias actualizaciones (v.g., si solo al final se cambia un "track" que contiene apuntadores a otros "tracks", el problema de alterar varios "tracks" es equivalente al problema de alterar un solo "track"). Así, todos los nodos en los cuales hay datos que actualiza la transacción van generando suficiente información para forzar atomicidad local (aún falta la atomicidad distribuida sobre nodos).

El algoritmo realmente se invoca al final de la transacción. En este momento el nodo coordinador envía un mensaje " LISTO? " a los nodos relevantes. Si los nodos remotos han progresado hasta el punto de poder hacer su parte atómicamente , contestan con el mensaje "OK". Si todos contestan ' OK ' el coordinador puede contestarle a quien invoca la transacción que sí se harán efectivas las actualizaciones y envía un mensaje "HAGA" a todos los nodos relevantes. Si un nodo se cae (falla) antes de recibir "LISTO? " , no podrá contestar "OK" y las actualizaciones no se harán. Si el coordinador se cae antes de recibir todos los mensajes "OK" , tampoco se harán . Si un nodo se cae después de contestar "OK" y el coordinador recibe todos los "OK" de todos modos se hará la actualización ya que eventualmente cuando se reinicie el coordinador sabrá que el mensaje "LISTO? " no fué procesado. Una descripción gráfica de este proceso se ilustra en el apéndice 4.

Ahora bien, el punto para recalcar en esta discusión es que tanto la descripción anterior dada en español como la gráfica del apéndice 4, son aún bastante informales (inclusive al leer toda la descripción en (LAMP79) queda la duda si se consideran todas las posibilidades).

En la figura1 se describe el algoritmo usando un lenguaje de programación imaginario para posteriormente ilustrar problemas específicos que se presentan al programar este tipo de algoritmos . (Para efectos de esta discusión es suficiente escribir el código del coordinador).

5.3 ALGUNOS PROBLEMAS QUE ILUSTRA EL EJEMPLO

El pedazo de programa presentado en la figura . ilustra una serie de problemas de descripción en algoritmos distribuidos que son difíciles de expresar en lenguajes convencionales.

```
Coordinador: Procedure;
      /*primera fase*/
      Todobien <-True
       Do destino ={ciudad1, ciudad2, ciudad3}  while (todobien)
          send ' LISTO? '  to destino
          if respuesta ≠ ' OK '  then todobien  <- false
       end
       /*segunda fase*/
       if todobien then /*caso en que se hacen
                          las actualizaciones*/
                     Do destino = {ciudad1, ciudad2, ciudad3}
                        send ' HAGA '  to destino
                        if timeout then retransmita;
                     end
                     else/* caso en que no se hacen
                             las actualizaciones */
                     Do destino = {ciudad1, ciudad2, ciudad3}
                        send ' DESHAGA '  to destino
                        if timeout then retransmita
                     end
end Coordinador,
```

Figura 1. Parte de un Programa Distribuido

Consideremos el problema de describir el modo de comunicación. Tal como está escrito el programa la primitiva de comunicación SEND está enviando el mismo mensaje varias veces, y peor aún, se envía solo cuando se recibe respuesta del envio anterior. Idealmente sería deseable poder enviar un solo mensaje a un grupo de destinos. Sea cual fuere la semántica y sintaxis, debe haber una estrecha relación al implantar el mecanismo con los protocolos de comunicación entre procesos provistos por el software de comunicación (la eficiencia a su vez depende de la capacidad de los algoritmos de enrutamiento de la red de enviar eficientemente mensajes a múltiples destinos).

Otro problema al definir la primitiva de envio de mensajes es la definición de cuándo acaba de ejecutar. Por ejemplo un SEND podrá acabar cuando el protocolo justamente envíe el mensaje, o cuando el protocolo reciba confirmación de la recepción del mensaje (cuál sería este equivalente cuando existe un grupo distinto?), o cuando se reciba respuesta (o sea incluye procesamiento) del destino. El primer tipo de solución implica crear mecanismos en el lenguaje para recibir respuestas asincrónicamente. El tercer tipo de solución puede limitar el paralelismo. El segundo tipo es similar al primero.

Adicionalmente al problema anterior, la comunicación debe definir a dónde se envía un mensaje: a un nodo?, a un programa que puede tener varios procesos?, a un proceso? (Nótese que este problema es similar al de cómo y dónde se reciben los mensajes). Al tratar de solucionar este problema se interna el diseñador en el problema de nombres que se exportan a otros nodos. En el ejemplo se aprecia este problema: "ciudad1" es un nombre a donde se envía un mensaje y que lo conoce tanto el "coordinador" como el proceso remoto. Qué es el mínimo que se debe exportar y cómo se expresa?

Este problema de comunicación se relacionan directamente con el problema de manejo de tiempo. En el ejemplo la construcción "if timeout ..." es bastante ambigua. Primero se debe definir en qué momento se inicia el contador del tiempo y seguramente es importante que el programador tenga control sobre la longitud del intervalo de tiempo. Segundo, se debe definir si el contador va a medir la duración del tiempo para que el mensaje se transmita al otro lado o si la duración es la del tiempo para recibir una respuesta (lo cual incluye el tiempo de procesar remotamente el mensaje enviado). En cualquier caso es claro que si el contador expira, se debe procesar a nivel de lenguaje como una interrupción asincrónica (similar a la recepción de mensajes).

Finalmente, en el ejemplo no se han expresado condiciones para reiniciar el programa en el caso de fallas. Recordemos que este algoritmo es tal que si ocurre una falla durante la primera fase (v. g., se va la luz y se borra la memoria principal), al reiniciar el coordinador simplemente se olvida de lo hecho anteriormente. Sin embargo, si la falla ocurre durante la segunda fase, al reiniciar debe en principio repetir la segunda fase. Cómo expresa el programador éste tipo de lógica ? Cuál sería un "buen" mecanismo (semántica y sintaxis) para expresar este tipo de situaciones?

5.4 VISTAZO GLOBAL A ALGUNAS PROPUESTAS

Se pretende en esta sección presentar muy brevemente conceptos de algunos de los principales trabajos relacionados con programación distribuida. Es importante anotar que estos proyectos son recientes y por lo tanto en la actualidad seguramente han evolucionado mucho más de lo reportado en las referencias.

PLITS (FELD79) es un proyecto desarrollado en la Universidad de Rochester que desde el punto de vista de programación distribuída introduce varios conceptos. La noción de módulo se usa para aislar todo pedazo de código donde puede existir concurrencia, donde se pueden compartir objetos , y donde inclusive se puede programar en un lenguaje diferente al de otros módulos. Todos los módulos se comunican vía mensajes. El concepto de mensaje es introducido como un nuevo tipo de datos; cada módulo de componer, enviar, recibir, y descomponer los mensajes; y existen primitivas para manipular los mensajes (v. g., put, remove, absent, present). Los mensajes se envían a nombres de módulos y se reciben dentro de un módulo en un área común.

MOD (COOK79) es un lenguaje para programación distribuida desarrollado en la Universidad de Wisconsin . Es un descendente de MODULA y contiene las nociones de módulos y redes de módulos. Sin embargo la comunicación se hace por invocación remota y no por envío directo de mensajes. Los mecanismos de sincronización son los convencionales de sistemas concurrentes y no se propone ningún mecanismo de confiabilidad en el lenguaje mismo.

Existe una propuesta de extensión a CLU (LISK79) en la cual se intercambian mensajes explícitamente y se definen puertos para manipular a nivel de lenguaje. Sin embargo los mecanismos de confiabilidad y sincronización son los estándares para sistemas concurrentes.

Finalmente existen propuestas con modelos de computación distribuídos pero poco realistas como en (HOAR78, BRIN78)

6. CONCLUSIONES

Las reflexiones anteriores ilustran varios puntos importantes:

1) La programación concurrente es clave para aumentar la eficiencia de ciertos sistemas, mientras que la programación distribuida es necesaria para implantar algoritmos distribuidos.

2) Aunque se ha avanzado bastante en resolver problemas fundamentales en programación concurrente, estas soluciones se aplican poco a resolver los problemas fundamentales en programación distribuida (después de todo usan modelos de computación bastante diferentes).

3) Es fundamental usar "buenos" lenguajes ya que el problema no es si se puede o no implantar algoritmos concurrentes o distribuidos, sino usar herramientas que se entiendan, que sean flexibles, que sean potentes, que ayuden a disminuír la probabilidad de error y confusión del programador, etc.

4) Aunque "nuestra instalación" no posea lenguajes distribuídos (o inclusive concurrentes), estas herramientas se pueden utilizar en la actualidad como un paso adicional de desarrollo de software antes de traducir a lo convencional.

5) El area de programación distribuida está aún por explorar y lo más inquietante es que tendrá que explorarse a fondo si queremos explotar realmente las arquitecturas distribuidas que son hoy en día una realidad comercial!

REFERENCIAS

(ANDL77) Andler, S., "Synchronization Primitives and the Verification of Concurrent Programs," Carnegie-Mellon University, Mayo 1977

(BLOO79) Bloom, T., "Synchronization Mechanisms for Modular Programming Languages.", MIT/LCS/TR-211, Enero1979

(BRIN70) Brinch Hansen, P. , " The Nucleus of a Multiprogramming System" , CACM , Abril 1970, pp. 238-241

(BRIN78) Brinch Hansen, P. , " Distributed Processes: A Concurrent Programming Concept " , CACM , Nov 1978, pp. 934-941.

(BRYA78) Bryant, R. E. y Dennis, J. B., " Concurrent Programming ,MIT/LCS/TM-115, Octubre 1978

(CAMP74) Campbell, R. H. y Habermann, A. N. , " The Specification of Process Synchronization by Path Expressions, " , Lecture Notes in Computer Science 16, Springer-Verlag, 1974

(COOK79) Cook, R. P. , " MOD-A Language for Distributed Processing ", the Ist Intl. Conference on Distributed Computing Systems, IEEE, Octubre 1979, pp. 233-241

(DAVI79) Davies, D., et. al., Computer Networks and their Protocols, John Wiley & Sons, 1979

(DIJK68) Dijkstra, E. W., " Cooperating Sequential Processes" Programming Languajes (F. Genuys, Ed), Academic Press, N. Y. 1968

(ELLI77) Ellis, C. A. , "A Robust Algorithm for Updating Duplicated Databases", Proc. Second Berkeley Workshop on Distributed Data Management and Computer Networks, Mayo 1977, pp. 146-158

(FELD79) Feldman, J. A, " High level Programming for Distributed Computing " , CACM , Junio 1979 pp. 353-368

(GRAY78) Gray, J., " Notes on Data Base Operating Systems", IBM Research Lab., San José, RJ 2188(30001), Febrero 1978

(HOAR74) Hoare, C.A.R., " Monitors: An Operating Systems Structuring Concept, " , CACM Octubre 1974, pp. 549-557

(HOAR78) Hoare, C.A.R., " Communicating Sequential Processes" , CACM , Agosto 1978, pp. 666-677

(KANE79) Kaneko, A., et. al., " Logical Clock Synchronization Method for Duplicated Database Control" , The Ist Intl. Conference on Distributed Computing Systems, IEEE, Octubre 1979, pp 601-611

(LAMP79) Lampson , B. y Sturgis, H., " Crash Recovery in a Distributed Data Storage System", Xerox PARC, Marzo 1979 (aparecerá en CACM)

(LAMP80) Lampson, B.W. y Redell, D.D., " Experience with Processes and Monitors in Mesa, CACM , Febrero 1980, pp. 105-117

(LIND79) Lindsay, B.G., et. al., "Notes on Distributed Databases" , IBM Research Lab., San José RJ2571(33471), Julio 1979

(LISK79) Liskov, B., " Primitives for Distributed Computing" Proc. of 7th Symposium on Operating Systems Principles Diciembre 1979, pp33-42

(MAO80) Mao, T.W., y Yeh, R.T., " Communication Port: A Language Concept for Concurrent Programming", IEEE Transactions on Software Engineering, Marzo 1980, pp 194-204

(REED78) Reed, D. "Naming and Synchronization in a Decentralized Computer System ", MIT/LCS/TR-205, Octubre 1978

(THOM79) Thomas, R.H., " A Majority Consensus Approach to Concurrency Control ", TODS , Junio 1979, pp 180-209