Sobre administradores de recursos redundantes en sistemas distribuídos.

Claudio A. Rada Juan E. Sicardi[1] Jorge R. Ardenghi[2]
Departamento de Ciencias de la Computación
Universidad Nacional del Sur
Avda. Alem 1253
Bahía Blanca
Argentina

Resumen

La tolerancia a las fallas es una característica importante que presentan los sistemas distribuidos. Uno de los mecanismos mas utilizados para ello es la replicación, sea de datos, de procesamiento o de comunicación. La replicación de datos ha sido una de las más estudiadas por lo que ofrece en cuanto a disponibilidad y rendimiento. La replicación de comunicación se ha utilizado para proveer servicios confiables de mensajes. Sin embargo es poco lo que se ha hecho en cuanto a la replicación de procesamiento.

Por ello este trabajo propone un diseño de administradores de procesamiento redundante en un ambiente de sistemas distribuidos

Palabras Claves: Sistemas Distribuidos. Tolerancia a las Fallas. Protocolos de Control de Réplicas. Procesamiento Redundante. Votación. Consenso. Recuperación.

[1] e-mail: csicardi@criba.edu.ar
[2] Instituto de Ciencias e Ingeniería en Computación e-mail: jra@criba.edu.ar

Sobre administradores de recursos redundantes en sistemas distribuídos.

Introducción.

Los sistemas de computación distribuidos se caracterizan por brindar una gran cantidad de recursos computacionales y de almacenamiento. Dichos recursos deben ser explotados adecuadamente para construir sistemas de alta disponibilidad, performance y confiabilidad.

Una de las características más importantes que debe poseer un sistema distribuido, es su capacidad de *tolerancia a las fallas*, sin degradar la performance de forma significativa. Un concepto que surge inherentemente del estudio de la tolerancia a las fallas es el de *replicación* (multiplicación de componentes y recursos),

Es de interés distinguir tres clases de replicación:

. de datos.
. de cómputo o procesamiento.
. de comunicación.

A lo largo del trabajo presente, se analizan los distintos problemas que se presentan en la replicación de datos, para extender sus soluciones al tratamiento de la redundancia de procesos. El ejemplo se presenta similar a un protocolo de control de réplicas, donde las mismas son comparables con procesos redundantes. Se basa en la plataforma de un sistema distribuido asincrónico, en el cual los procesos se comunican por mensajes y aunque el medio de comunicación se supone confiable, se prevee la posibilidad de fallas.

Características de la replicación de datos

La mayor parte del trabajo en sistemas de datos replicados se ha realizado en el contexto de bases de datos distribuidas, que utilizan *transacciones* para modelar los requerimientos de acceso. Una transacción consiste en un conjunto de operaciones relacionadas que leen y posiblemente actualizan el estado de una base de datos, teniendo en cuenta las propiedades de atomicidad, permanencia y serilizabilidad.

El uso del concepto de transacción para modelar computaciones distribuidas provee un medio conveniente para resolver control de concurrencia y problemas de manejo de redundancia.

Protocolos de Control de Réplicas.

Un Protocolo de Control de Réplicas es un nivel de sincronización impuesto por un sistema distribuido para la transparencia de replicación presentando a los usuarios la ilusión de una copia única de los ítems de datos. Este protocolo provee un conjunto de reglas para regular la lectura y escritura de las réplicas y para determinar los valores de los datos.

Básicamente, el comportamiento para asegurar el control de las réplicas es el siguiente:

"Leer de la réplica local, pero escribir en todas las réplicas."

Este es el metodo de control *read-one/write-all* , que obviamente mantiene la consistencia de los datos, y aunque mantenga una alta disponibilidad para la lectura de datos, la performance de las operaciones de escritura cae estrepitosamente, porque la posibilidad de que todas las copias estén disponibles para actualización es relativamente baja. En resumen, la atomicidad reduce la performance.

Sin embargo, no todas las réplicas necesitan ser actualizadas por una operación de escritura. Se puede considerar el siguiente protocolo:

"Las operaciones de lectura y escritura acceden una mayoría de las réplicas."

Ya que una operación de escritura instala el nuevo valor en una mayoría de las réplicas, y cualquier par de mayorías tienen al menos una copia en común, se garantiza que para una mayoría-de-lectura, esta contenga el valor corriente. Es más, una operación de lectura y una de escritura, o dos operaciones de escritura en el mismo ítem de datos no se ejecutarán concurrentemente, porque deben acceder a una réplica en común. El nodo que administra la réplica en común fuerza la sincronización correcta, de tal forma que esta es modificada por una única transacción por vez. Este es un caso especial del método de consenso por quorum ("Quorum consensus"). La propiedad mas importante del "consenso por mayoría" es que para dos decisiones por mayoría cualesquiera, al menos un servidor ha participado de ambas.

En el primero de los métodos, la alta disponibilidad para lecturas no es despreciable. En el segundo, la performance ante las escrituras mejora ampliamente. Una consideración de compromiso entre ambas sería: asegurar el acceso exclusivo a todas las réplicas, y aceptar la escritura si es posible realizarla en una mayoría de ellas. Esto garantiza escritura uniforme, evitando la degradación del sistema por falla en alguno de sus nodos o módulos.

Vemos que, como consecuencia de su redundancia inherente, los sistemas distribuidos proveen una manera efectiva de aplicar técnicas de tolerancia a las fallas. A medida que la complejidad de los sistemas crece, aumenta la probabilidad de fallas en sus componentes, la cual puede afectar adversamente la performance y utilidad de los mismos. Por lo tanto la confiabilidad, la disponibilidad y la tolerancia a las fallas resultan ser importantes puntos de diseño en los sistemas distribuidos. En un entorno de computación como éste , los procesos que interactúan lo hacen a través de mensajes que atraviesan varios niveles de software, y consecuentemente es entendible que aplicar técnicas de tolerancia a las fallas pueda ser muy costoso en términos de tiempo de ejecución.

En el presente trabajo, se emplea una técnica de diseño de software tolerante a las fallas, y se identifican las características de los módulos que soportan la implementación de diferentes métodos de administración de archivos replicados. Esto permite desarrollar una capa de software sobre el sistema operativo, creando un entorno para la aplicación de técnicas de tolerancia a las fallas.

Consideraciones de Diseño.

Durante el diseño de un sistema distribuido deben tenerse en cuenta dos problemas sobresalientes:

- Control de concurrencia: incluye planificación de la ejecución concurrente de tareas. Requisito de serializabilidad.

- Administración de los recursos redundantes: preservación de la consistencia entre los recursos replicados, y mantenimiento de la información necesaria para soportar recuperación.

Las *transacciones* son un paradigma confiable para la realización de aplicaciones distribuidas. Las técnicas para manejar la redundancia y mantener la consistencia de los objetos replicados, se caracterizan en algoritmos de control *centralizados* y *descentralizados*. El algoritmo de control centralizado soporta fuertes requerimientos de consistencia, pero es susceptible de puntos de falla únicos. Los algoritmos descentralizados soportan requerimientos de consistencia "mas débiles", y puede por lo tanto incrementar el throughput del sistema, pero en algunos casos pueden presentar n puntos de falla. El manejo y control de redundancias (tradicionalmente algoritmos de voting), no sólo se usa para mantener la consistencia de los recursos replicados, sino también es responsable de la recuperación del sistema en presencia de caídas de los nodos y fallas en las comunicaciones.

Una técnica que provee soporte para Tolerancia a las Fallas.

Por *tolerancia a las fallas* entendemos a la habilidad del sistema de continuar su ejecución a pesar de la ocurrencia de fallas o errores.

Es necesario identificar las características de los módulos de programa para facilitar la implementación de los algoritmos de tolerancia a las fallas. Veamos la organización propuesta para los sistemas distribuidos, más los algoritmos y protocolos. Se asume la arquitectura de los modulos de cómputo soportando *broadcast* confiable, recuperación propia, tolerancia a las fallas selectiva, y memoria permanente.

Organización del Control de Recursos Redundantes (CRR): El diseño base comprende un conjunto de módulos de computo ("nodo"). Estos módulos (servidores/recursos) se comunican e interactúan con un *coordinador de nodo (CN)* a través de mensajes (datagramas). Un conjunto de nodos forma un *cluster*. Varios clusters conviven simultáneamente en el sistema, independientes entre si, aunque bajo supervisión de un *coordinador de clusters (CC)*.

Diseño de un nodo: Cada nodo tiene varios elementos de procesamiento que pueden ser configurados dinámicamente para ser utilizados como un modulos de cómputo redundante o como un módulo independiente. El número de elementos de procesamiento depende de los requerimientos de confiabilidad y de tolerancia, que se representan según el grado de replicación efectuada. Un nodo tiene dos modos de operación: tolerante a las fallas y multiproceso. Para tareas *no críticas* el nodo actúa en modo multiproceso, mientras que para tareas *críticas* los diferentes servidores ejecutan la misma tarea sincrónicamente y usan un procedimiento de voto para enmascarar los efectos de los módulos que fallan. Aunque es posible elaborar un proceso de selección de nodo *coordinador*, este se selecciona de antemano, y desde el comienzo es parte del conjunto de procesos libres de fallas. Mas adelante se contempla la posibilidad de compensación de falla del coordinador. Por lo tanto inicialmente no hay un proceso de seleccion de coordinador, sino que este es predefinido.

El proceso de actualización de copias se cumple atómicamente de acuerdo al procedimiento de comunicación por "broadcast" de requerimientos descripto por Thomas [Tho79]: El servidor (en este caso el coordinador de nodo), que recibe el requerimiento de un cliente, hace un *broadcast* del requerimiento a los otros servidores, quienes devuelven a éste los resultados.

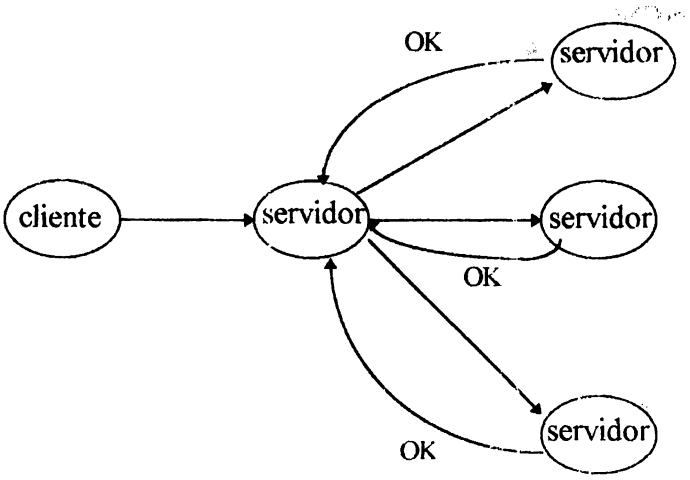


Figura 1: Broadcasting

Ejemplo: Si la necesidad de acceder exclusivamente a todas las copias de un recurso para ejecutar la tarea es satisfecha, aún no se asegura que todos los servidores del recurso la terminen satisfactoriamente (y menos aún, que todos arrojen los mismos resultados). Es necesario, si se obtiene una cantidad parcial de los resultados, que se sepa cómo evaluarlos, y si se puede satisfacer el requerimiento. Es aquí que vale hacer una analogía con el ejemplo anterior de Protocolos de Control de Réplicas (*read-one/write-all* y *majority consensus*): sobre el parcial o total de los resultados, se toma el valor válido sobre la mayoría coincidente -si existe-.

Coordinador de Cluster : definimos un "cluster" como una colección de módulos de cómputo en una red, que pueden funcionar como un recurso único a través del uso de software de administración extra. Para cada conjunto de nodos que forman un cluster, existe un nodo diseñado especialmente como coordinador del cluster. Sus tareas son las de supervisar los procedimientos de recuperación de cualquier coordinador de nodo (cuando éste está caído), o la de reintentar una operación en otro nodo cuando aquel que estaba a cargo originalmente haya fallado.

Tres operaciones distinguen su actividad:

1. Escuchar continuamente mensajes de status de los coord. de nodo CN(i):
   Mientras (.T.)
     recv_msg(CN(i), status)
     En caso que:
        Recibe un mensaje ERROR_CN(i)
              - Recuperar la cola de tareas de CN(i)
              - Si fué posible, distribuir las tareas en los otros nodos.
              - Algoritmo de restitución del CN(i).
        Recibe mensaje de status distinto de ERROR_CN(i)
              - Actualizar el timer de espera de status del CN(i).
     Fin_Casos
   Fin_Mientras

2. Si un timeout indica que hace mucho que no se reciben mensajes de CN(i):
   Si TimeOut(CN(i))

```
- Enviar_req(CN(i), status)    /* Enviar un pedido de status.*/
- Si (respuesta = ERROR_CN(i)) .or. (timeout(rta))
  /* Si no se recibe respuesta, o respuesta = ERROR_CN(i) */
        - Levantar un nuevo coordinador de nodo CN(i)'.
        - Conectarlo a los servidores de CN(i)'.
        - Asignarle los pedidos pendientes -si quedara alguno-.
  Fin_Si
Fin_Si
```

3. Operaciones de control de "supervivencia" del coordinador de clusters.
   Obviamente, dejar el CC como un módulo central de todo el sistema, es un punto de falla único. Se debe contemplar un algoritmo de mantenimiento/recuperación del mismo.

Capacidad de Tolerancia a las Fallas Selectiva: la redundancia, junto con algoritmos de tolerancia a las fallas, se usan para asegurar la ejecución atómica de tareas críticas y la recuperación del sistema cuando ocurren errores. Como las operaciones críticas constituyen solamente una parte de todas las operaciones, el sistema debe ser lo suficientemente inteligente como para detectar la necesidad o no de operar en el modo tolerante a las fallas. Sin embargo, para que un nodo actúe en dos modos se deben proveer técnicas para su reconfiguración.

Configuraciones de un nodo: El soporte para posibilitar dos modos de procesamiento puede ser provisto por semáforos. Como capa de software de control, las tareas son descriptas por grafos, cuyos nodos representan las estructuras computacionales y cuyos arcos representan las restricciones de dependencias. Las tareas críticas se distinguen de las no críticas de acuerdo a su semántica en el cómputo. El coordinador, con la ayuda de semáforos, mantiene una cola de tareas listas para ser ejecutadas concurrentemente tan pronto como un recurso esté disponible. El manejador de tal cola, planifica las tareas de forma primero-en-llegar-primero-en-salir (FIFO). Pero obsérvese que la planificación debería considerar que la ejecución de tareas críticas requiere de todos los servidores en un nodo.

Es posible utilizar dos esquemas de planificación de las tareas críticas. La primera de ellas utiliza *planificación apropiativa*: las tareas críticas se apropian de todas las demás; ante una tarea crítica, quien las monitorea se apropia de todas las otras. En el segundo esquema, cualquier tarea crítica debe esperar que todas las demás se completen, y no se planifica tarea alguna durante este período.

Voto Distribuido: Haciendo un análisis, un resultado erróneo se asume como consecuencia de haber producido datos corruptos, como ausencia de datos o, normalmente, por un resultado de "ERROR" enviado por un servidor al coordinador.
Para el caso del ejemplo, los resultados de los procesos asumen dos estados: OK o RECHAZAR. Cualquier resultado distinto se asume como "rechazar". Durante el modo de operación tolerante a las fallas (tareas críticas), el proceso coordinador asociado a un nodo, supervisa el algoritmo de voto distribuido y el commit de la ejecucion de la tarea. No hay selección de coordinador alguno, sino que está predeterminado en el levantamiento (boot) del nodo. Sin embargo hay un algoritmo de recuperación que actúa en caso de falla o caída del coordinador. La detección de tales fallas las hace el coordinador de clusters que esté activo en ese momento.

A continuación se describe el algoritmo de voto y los procedimientos relacionados para un voto distribuido en un ambiente tolerante a las fallas

Tarea Crítica:

Fase 1

```
/* El coordinador de nodo recibe un requerimiento de operación crítica. */
CN(i): recv_msg (operación)
Si tipo_operación = "crítica"
   /* El requerimiento es enviado a c/u de sus servidores asociados */
   ∀j enviar_req (servidor(j),operación)
```

Fase 2

```
/* Cada uno de los servidores S(j) intenta la operación. */
∀j , S(j):
        - Intentar ejecutar la operación.
        - Evaluar Resultado.
        - Votar (OK/Rechazar) /* En función del resultado obtenido, envía un voto de
                                 aceptación o rechazo */
```

Fase 3:

```
/* El coordinador de nodo recibe los votos de los servidores: */
CN(i): recv_msg(S(i), voto)
En caso que:
        - Todos los votos coinciden:
                Se acepta el resultado (OK/RECHAZAR)
        - Solo mas de la mitad de los votos coinciden:
                Se acepta el resultado de la mayoría como válido.
                + Si resultado válido es OK:
                        /* Ver qué sucede con aquellos que votaron RECHAZAR. */
                        ∀j tal que S(j) votó RECHAZAR:
                                Chequear_status(S(j))
                                si (status = "falla")
                                        Recuperar S(j)
                                fin_si
                  fin_si
                + Si resultado válido es RECHAZAR
                        /* Si es necesario, deshacer / ignorar los resultados de los que
                           votaron OK. */
        - No hay mayoría de resultados, pero algunos servidores emitieron su voto:
                Verificar si el coordinador está funcionando bien:
                + En caso afirmativo:
                        /* Enviar RECHAZAR al coordinador de clusters. */
                        enviar_req(CN(i), RECHAZAR)
                        /* Verificar si los servidores están OK. */
                        ∀j tal que S(j) votó RECHAZAR:
                                Chequear_status(S(j))
                                si (status = "falla")
```

Recuperar_Servidor(j)
fin_si
+ En caso negativo:
/* Enviar pedido de recuperación del Coordinador de Nodo al Coordinador de Cluster. */
CN(i): enviar_req(CC, recuperar)
- Nadie emite el voto (timeout/el coord. de nodo está aislado):
Recuperar el Coordinador de Nodo.

Cada proceso servidor computa un resultado, que debe ser votado antes de autorizarse el "commit". Las funciones usadas para el cómputo de estos resultados dependen de las transacciones a aplicar. En nuestro ejemplo del consenso por mayoría, la función que habilitaría una operación de escritura, sería la de obtener una mayoría de bloqueos del ítem de dato a (sobre)escribir. - En el caso general, el consenso se obtendría en función de la cantidad de accesos al recurso/proceso en forma exclusiva logrados -. Una vez conseguida la mayoría, en una segunda fase, bastaría con enviar la orden de escritura a todos los procesos servidores. En caso de no poder obtenerse la mayoría -por cualquier razón X- es suficiente con retirar el requerimiento del sistema.

En el algoritmo, la fase 2 computa los resultados en cada servidor y se envían los mismos al coordinador. La fase 3 incluye el voto de los resultados y determina si la operación puede concluirse. Cada uno de los servidores recibe la acción determinada por el coordinador, y actúa de acuerdo a ello.

La decisión del coordinador está influenciada por la comparación con tres casos posibles:

1. Todos los valores coinciden. En este caso se dice que hay un "consenso completo" El coordinado hace un "broadcast" de los resultados a todos los servidores. Cada uno completa la operación y envia un OK como respuesta.

2. Solo se obtiene una mayoría de valores que coinciden, y algunos de los resultados son dispares. El coordinador decide estar de acuerdo con la mayoría ("consenso democrático"). En este caso hay dos grupos de servidores, aquellos que pertenecen a la mayoría y aquellos que no. Usando el resultado de la mayoría como el correcto, se pide a todos los servidores que acepten tal resultado como válido, pero se necesita investigar en qué condiciones se encuentran aquellos procesos servidores que han "fallado" (recuperación de servidor).

3. No se obtiene mayoría alguna. Esto produce un resultado de RECHAZAR por parte del coordinador, y obviamente es necesario disparar un proceso de diagnóstico que obligue saber cuántos servidores están en estado "ok" como para llevar adelante la operación. Si ninguno de los servidores está "ok" se considera que es el coordinador de nodo quien ha caído. Esto requiere recuperación externa e interviene el coordinador de cluster. Si el coordinador de nodo es quien falla, en este caso, se corre el algoritmo de recuperación de coordinador. Si solo algunos de los servidores están "ok", se considera este subconjunto del conjunto original de servidores como el conjunto de voto, y se realiza de nuevo la evaluación de consenso.

Conclusiones.

El uso de clusters para la computación distribuida es ya popular con la proliferación de las estaciones de trabajo poderosas y de bajo costo. Sin embargo un grupo de estaciones interconectadas no constituyen un cluster, a menos que posean software adecuado de administración. Por otra parte, el

diseño de cualquier sistema distribuido es complicado, a causa de la heterogeneidad de sus componentes, los problemas de comunicación, la necesidad de sincronización de los procesos cooperativos, y el mantenimiento de la consistencia entre las copias múltiples de información. El objetivo fue aprovechar los conceptos y métodos utilizados respecto al último de los problemas mencionados, en busca de un protocolo sencillo y general de administración de recursos redundantes, apuntando a sacar ventajas en tolerancia a las fallas, ejecución de tareas concurrentes en varios módulos de cómputo, y compartimiento de los recursos que nos brindan los sistemas distribuidos.

Los módulos propuestos cubren la implementación de algoritmos para el manejo o administración de la redundancia, orientado especialmente a procesos. La recuperación ante fallas en los nodos se hace con diagnóstico local, dentro de cada nodo, siendo de complejidad reducida y relativamente eficiente.

Además, es necesario poder afirmar que lo ganado en tolerancia, no genere demasiado tráfico de mensajes extra. La jerarquía de dos niveles impuesta (clusters-nodos), asegura un porcentaje muy alto de disponibilidad y confiabilidad. Un análisis mas detallado indicaría que con un orden de replicación no mayor de cuatro, se logra dicha alta disponibilidad, sin afectar demasiado la sobrecarga ni el tráfico de red. Si también consideramos que el voto por mayoría funciona mejor sobre un conjunto impar de votantes, concluímos que con tres o cinco réplicas se satisfacen los requisitos de equilibrio entre la manera de asegurar la detección temprana de fallas, sin enviar demasiados mensajes que determinen qué módulos de procesamiento están aún con vida.

Es oportuno destacar que el entorno de red tiene un impacto muy importante en el uso efectivo de la replicación. El uso de broadcast y multicast a veces es muy caro, o no es siempre posible. La replicación de mensajes se ha determinado como una solución general a problemas como, por ejemplo, de los Generales Bizantinos.

Cualquier forma de implementación de lo desarrollado, no puede dejar de considerar criterios indispensables a la hora de la evaluación de lo producido: soporte heterogéneo; soporte de procesamiento paralelo; comunicación por mensajes; balanceo de la carga; ausencia de un punto de falla único; transparencia al usuario y adaptación a los cambios dinámicos.

Por último, es conveniente comentar que este estudio se ha desarrollado en forma paralela a la implementación, por parte de los autores, de una plataforma de administración de la replicación que, aunque simple, respetara las condiciones vistas y tuviera por objetivo facilitar y acompañar los resultados aquí expuestos. La misma se ha desarrollado en lenguaje C, corriendo inicialmente sobre el sistema operativo Linux, y extendida luego a Unix, respetando el standard IEEE Posix.1, y la implementación 4.3+BSD. En el diagrama anexo, se muestra la interacción entre los módulos de cómputos pertenecientes a un nodo atendiendo dos requerimientos de operaciones críticas independientes. Los procesos servidores se muestran como burbujas sombreadas, mientras que aquellas sin sombreo alguno, son procesos creados por las anteriores para atención del servicio. Nótese la relación estrecha que existe entre el nivel de tolerancia, el grado de replicación y la cantidad de mensajes necesaria para comunicación.

Bibliografía.

[Har92] S. Hariri, A. Choudhary, B. Sarikaya. *Architectural Support for Designing Fault-Tolerant Open Distributed Systems.* IEEE Computer.

[Kap94] J. Kaplan, M. Nelson. *A Comparision of Queueing, Cluster and Distributed Computing Systems*. NASA Langley Research Center.

[Cou89] G. Coulouris, J. Dollimore. *Distributed Systems. Concepts and Design*. Addison-Wesley. 1989.

[Ba92] Jean Bacon. *Concurrent Systems. An Integrated Approach to Operating Systems, Database, and Distributed Systems*. Addison Wesley. 1992.

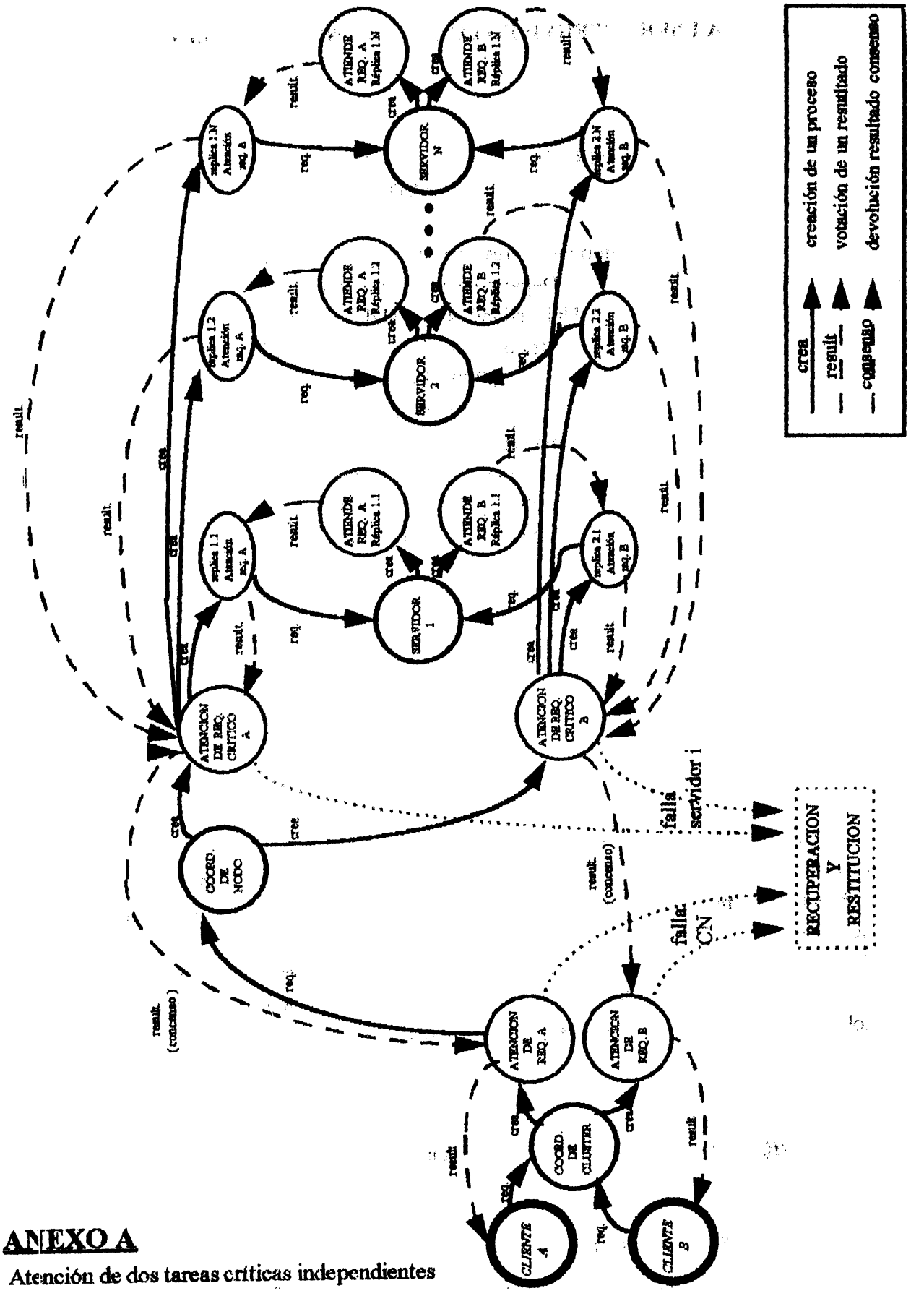
[Cas94] L. Casavant, M. Singhal. *Distributed Computing Systems.*

[Slo94] Morris Sloman. *Network and Distributed Systems Management.* Addison Wesley (62745).

[Mo94] L. Moser, P. Melliar-Smith, V. Agrawala. *Processor Membership in Asynchronous Distributed Systems.* IEEE Transactions on Paralell and Distributed Systems.

[Tho79] R.H. Thomas. *A majority consensus approach to concurrency control in multiple copy databases.* ACM Trans. on Database Systems, vol. 4 n. 2

[Ste93] W. Stevens. *Advanced Programming in the UNIX Environment.* Addison-Wesley (56317).



ANEXO A

Atención de dos tareas críticas independientes