La tolerancia a fallas en los sistemas distribuidos.

Roberto Gómez Cárdenas
ITESM-CEM
rogomez@campus.cem.itesm.mx

Noviembre 1996

Resumen

Una de las ventajas más importantes dentro de los sistemas distribuidos es la tolerancia a fallas. Sin embargo las técnicas para diseñar algoritmos tolerantes a fallas no son simples. El objetivo de este documento es el de dar a conocer la terminología usada en la tolerancia a fallas. También se presentan algunas técnicas a nivel software para el diseño de algortimos distribuidos robustos. Las siguientes notas son parte de un compendio de apuntes que se esta escribiendo sobre el área de algoritmos distribuidos. Cualquier recomendación o comentario se agradecerá hacerlo llegar al autor.

Los algoritmos distribuidos son ejecutados en sistemas formados por entidades, (procesos, procesadores, computadoras, redes, ect), ligadas entre ellas por vías de comunicación. Por comodidad a lo largo de este documento nos referiremos a estas identidades como *sitios*. Un sitio puede estar compuesto de uno o varios procesos, de al menos un nivel de memoria, y puede disponer de diferentes dispositivos. El riesgo de que un sitio tenga una falla o que una línea de comunicación sea cortada es considerable trayendo consecuencias negativas, a veces graves, en el funcionamiento del sistema en general. Es importante notar que no se puede predecir el momento donde la falla se producirá, debido a que la falla es de esencia aleatoria.

Una de las propiedades de los algoritmos distribuidos es su tolerancia a las fallas. Un algoritmo debe ser capaz de trabajar aunque un sitio o una línea de comunicación tenga fallas. Los algoritmos presentados en los capítulos precedentes no fueron definidos en función de tal objetivo y su comportamiento es imprevisible en presencia de falla. El objetivo del presente capítulo es presentar las definiciones concernientes al área de tolerancia a fallas dentro de los sistemas distribuidos.

En un principio, presentamos las definiciones de los diferentes conceptos implicados en el dominio de la tolerancia a las fallas. Una vez estas definiciones dadas, describimos las técnicas más utilizadas para la tolerancia a fallas.

Este capítulo esta dividido como sigue: la primera sección recuerda la terminología del dominio de la seguridad: definición de una falla, la diferencia entre ésta y una avería, una falta y un error. La segunda presenta las propiedades de los modelos cliente/servidor y transaccional. La tercera muestra las características de un sistema tolerante a las fallas. La cuarta sección presenta diferentes aspectos relacionados con redundancia. La quinta sección describe diversas técnicas utilizadas para la concepción de algoritmos distribuidos tolerantes a las fallas.

1 Definición de falta, error, falla

El uso de palabras como seguridad de funcionamiento, tratamiento de errores, faltas, fallas y averías se presta a confusión, ya que el uso no es el mismo para todos los autores. No existe una terminología en español de los diferentes conceptos asociados a lo que es la tolerancia de fallas. En [AnLe81] y [Flav83] los autores determinan las nociones de falta, error y falla, sin embargo adaptamos las definiciones dadas por Laprie y Deswarte en [Lap88] y [Desw90] respectivamente[1].

Ellos consideran la falla de un sistema como la falta a la especificación de servicio pedido. La especificación siendo una descripción autorizada del servicio esperado. Esta especificación debe

[1]Lo que en realidad se hizo fue una traducción de los diferentes terminos mencionados en dichas fuentes bibliográficas.

permitir que toda investigación sobre la causa de una falla concierna solamente la operación interna del sistema. El fallo ocurre cuando el sistema tiene un comportamiento erróneo.

El comportamiento de un sistema es la función que realiza, y lo que le permite llevar a cabo dicha función es su estructura. Una estructura esta dividida en estados, donde un estado es definido como la razón de ser de acuerdo a un un conjunto de circunstancias. Por lo tanto, para poder encontrar la causa de una falta debemos definir dos conceptos: *transición errónea* y *estado erróneo*.

Una transición errónea de un sistema es una transición de un estado interno del sistema a otro, que más tarde puede provacar una falta. Si la transición no es errónea, entonces es *valida*. Un estado erróneo es un estado interno que después de una secuencia de transiciones válidas, puede provocar una falta. Si un estado interno no es erróneo, entonces es válido.

Un error es la parte del estado del sistema (con relación a los procesos de tratamiento) que es susceptible de ocasionar un fallo. La causa supuesta del error es una falta. Un error es entonces la manifestación de una falta en el sistema, y una falla es entonces el efecto de un error sobre el servicio.

Una falta se dice activa cuando produce un error. Una falta es, ya sea una falta interna que estaba precedentemente durmiente (i.e. no producía error), y que ha sido activada por el proceso de tratamiento, sea una falta externa. Una falta interna puede pasar del estado durmiente al estado activo.

Un error es por naturaleza temporal. Puede estar latente o detectado: un error es latente mientras que no ha sido reconocido como tal; es detectado sea por los mecanismos de detección de error que analizan el estado del sistema, sea por el efecto del error sobre el servicio. Generalmente, un error propaga otros errores, nuevos, dentro de otras partes del sistema.

Si un sistema es considerado como un conjunto de componentes, la consecuencia del fallo de un componente es una falta interna por el sistema que lo contiene, y también una falta externa por el o los componenetes que interactúan con él.

En vista de comprender mejor la terminología tomamos el ejemplo siguiente: un progrmador que se equivoca tiene un fallo seguido a un error de razonamiento; la consecuencia es una falta durmiente en el software escrito. Cuando esta falta sea sensibilizada, se volverá activo. La falta activa produce una o varios errores en los datos tratados. Cuando estos errores afectan el servicio liberado, entonces se dice que el sistema falló.

Por otro lado una avería se define como el paro de funcionamiento accidental y momentáneo.

2 Clasificación de fallas

En [Stro84] Strong clasifica las fallas de la siguiente manera:

1. fallas francas;
2. fallas transitorias;
3. fallas temporales;
4. fallas bizantinas

2.1 Las fallas francas

En el caso de fallas francas, (failstop o crash en inglés), estas pueden presentarse en cualquier momento. En el instante mismo que un sitio tenga una falla, detiene definitivamente todas las operaciones que estaba haciendo, (a menos que sea reparado y reiniciado). Desde el punto de vista de vías de comunicación, la vía no será más considerada como operacional si es incapaz de transmitir algún mensaje. Desde el punto de vista de proceso, este interrumpe su trabajo y cesa de producir mensajes.

Ejemplos de este tipo de falla son la falla de un proceso, un interbloqueo en el sistema, o una ruptura de cable, así como una falla del controlador.

2.2 Las fallas transitorias

En este tipo de falla el proceso puede producir un resultado falso pero continua trabajando normalmente; un proceso puede perder un mensaje pero libera el resto de los mensajes que pasan por él. En este tipo de falla los resultados no son modificados, estan de acuerdo a la especificación pero pueden estar ausentes.

Encontramos como ejemplos de fallas transitorias: la pérdida de un mensaje, o la supresión de un proceso para evitar un interbloqueo. Este tipo de falla es conocida también bajo el nombre de falla intermitente o de omisión.

2.3 Las fallas temporales

Llamadas también fallas de timing o de desempeño. Estas fallas estan ligadas a la noción de tiempo y descansan sobre la existencia de una especificación precisa de comportamientos temporales. El mensaje llega antes o después del momento previsto.

El retardo de un mensaje, un reloj muy rápido, o la sobrecarga de un proceso que ocasiona un adelantamiento de vencimiento, son ejemplos de fallas temporales.

2.4 Las fallas bizantinas

En este grupo están reunidas todas las fallas no consideradas en las tres categorías anteriores. Son también conocidas bajo el nombre de fallas universales, o maliciosas. Todo comportamiento fuera de las especificaciones (principalmente cuando los resultados no estan de acuerdo con lo especificado), es calificado de comportamiento bizantino.

Un cambio en una política del orden en las transmisiones es una alteración de especificaciones y es, por lo tanto, una falla bizantina.

Dentro de esta categoría podemos encontrar una sub-clasificación la cual no es muy utilizada y se considera contravertida. Dentro de esta podemos distinguir:

1. **fallas bizantinas** NATURALES: error físico no detectado, o un error en el software.

2. **fallas bizantinas** MALICIOSAS: comportamiento dirigido a hacer suspender el sistema.

2.5 Jerarquización de fallas

Los cuatro tipos de fallas anteriores son jerarquizadas. Dicha jerarquización se encuentra descrita en la figura 1.

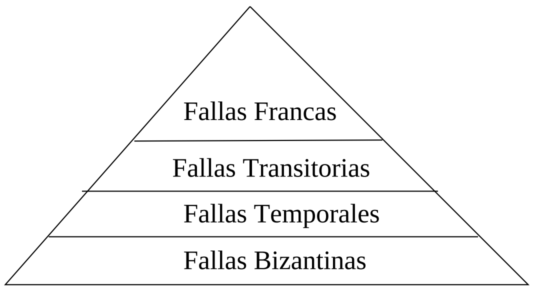

Figura 1: *Jerarquización de fallas.*

En el momento del diseño de de los diferentes componentes de un sistema tolerante a fallas , es común tomar en cuenta las hipótesis que involucran un solo tipo de falla, (que pertenece a uno de los tipos precedentes). El sistema es, entonces tolerante al tipo de falla escogida. Una de las hipótesis, más frecuentes es la tolerancia a las fallas de omisión.

3 El modelo cliente/servidor y transaccional

Uno de los modelos más usados en los sistemas distribuidos es el del clientte/servidor. Por lo tanto, antes de mostrar las características de tratamiento de fallas, presentamos las diferentes propiedades de estos dos modelos, así como su relación con el área de la tolerancia a fallas.

3.1 Las fallas en el modelo cliente/servidor

El modelo cliente/servidor ha sido descrito en la sección 1.2.3, aquí presentamos las posibles consecuencias posibles de una falla en ese modelo. La naturaleza del modelo conduce a considerar los dos casos siguientes de fallas:

1. **El servidor falló:** El cliente puede estar bloqueado indefinidamente esperando una respuesta a su demanda, a menos que una demora sea establecida. Sin embargo, si la demora escogida es demasiado pequeña, o si el tiempo de cálculo del servidor es demasiado lento, puede haber problemas, (¿cómo distinguir entre la caída de un servidor y un retardo largo).

2. **El cliente falló:** Cuando es el cliente el que falló, el servidor queda huerfano y no sabe a quien entregar el resultado de su trabajo, (el cual ha sido realizado en vano).

Tomemos un ejemplo, sea A un servidor y B un cliente, y supongamos que B pide a A consultar un valor sobre un archivo. Después de haber leído el valor, A debe saber cuando olvidarse de él. El problema es el siguiente: si B tiene una falla, ¿cuánto tiempo debe esperar A la confirmación de que B, el cliente, ha recibido bien el servico que pide?

Estos dos eventos (falla de cliente y de servidor) son detectables con una vigilancia mutua. A través de un intercambio de tiempo de espera el cliente (o el servidor) interroga su interlocutor que le responde. Así los dos saben que estan otra vez en funcionamiento.

En función de la naturaleza de las fallas observadas puede haber diferentes estrategias de recuperación, (retransmisión de mensajes, reejecucíón de servidores). La consecuencia mayor de estos mecanismos es que un servidor puede ser ejecutado varias veces, sin embargo, esto es inaceptable en ciertos casos. Si se trata, por ejemplo, de la lectura de un bloque específico de un fichero, esto no presenta algún problema; por el contrario, si el cliente pide una transferencia de un millón de pesos de una cuenta a otra, hay que razonar de manera diferente.

Este último ejemplo nos lleva a definir lo que es un *servidor idempotente*. Un servidor es idempotente cuando la repetida atención de un mismo servico solicitado por un cliente no afecta su estado interno. En el caso anterior la lectura de un bloque es una operación idempotenete mientras que la transferencia bancaria no.

3.2 El modelo transaccional

Una parte del vocabulario, y de las técnicas, del área de los algoritmos distribuidos proviene del área de las bases de datos distribuidas; en particular el concepto de *transacción*, (ver [Moss81, BeGo83, LSGG+79, BeHG87]

Una *transacción* se define como una secuencia de operaciones elementales que constituyen un todo. Estas operaciones son sometidas a condiciones de integridad, (i.e. se tienen asersiones a verificar en permanencia por estas condiciones). Una transacción puede terminarse de dos maneras diferentes: ya sea que fue *consumada* o fue *interrumpida*.

Una transacción fue *consumada* cuando termina con éxito; es decir, cuando cambia el estado del sistema de un estado consistente, hacia otro estado consistente. Una transacción es *interrumpida* cuando no fue consumada.

Otro concepto importante es el de *acción*. Una acción es una operación elemental considerada como ininterrumpible e insecable, particularmente en una transacción. Estas propiedades de insecabilidad y de ininterrumpibilidad inducen la definicion de *atomicidad*. Una acción es atómica cuando se ejecuta en su totalidad, o no se ejecuta nada, (es totalmente inoperante sobre el estado del sistema). Una acción presenta las propiedades de *totalidad* y de *seriabilidad*.

El concepto de *totalidad* está ligado al efecto de una acción sobre el estado final del sistema. Si una acción atómica termina con éxito, hace pasar el sistema de un estado a otro. Por el contrario, si la acción es interrumpida, el sistema se queda en el estado anterior a la acción, como si ninguna acción hubiera sido realizada.

La *seriabilidad* pide que el resultado de la ejecución concurrente de diversas operaciones sea la misma que aquella producida por la ejecución de sus mismas operaciones en un orden secuencial.

4 Características de un sistema tolerante a fallas

Una de las características principales de los sistemas distribuidos es su tolerancia a fallas. El objetivo de este capítulo es el presentar las técnicas relacionadas con los algoritmos distribuidos para lograr lo anterior. Ahora bien la implementación de un algoritmo distribuidos depende de la plataforma sobre la cual se vaya a implementar. En esta sección se presentan las principales características de un sistema tolerante a fallas, la mayor parte de esta sección esta basado en el libro de Anderson y Lee [AnLe81] si el lector desea profundizar más puede consultar dicha fuente.

Para poder certificar que un componente no funciona bien, primero se tiene que definir que es lo que ese componente debe de hacer. Esto implica que es necesario establecer la especificación del supuesto comportamiento del sistema.

Durante el diseño y la implementación de un sistema es posible que las especificaciones cambien. Una vez que se cuenta con una definición de como debe comportarse el sistema se tiene un standard para poder juzgar el funcionamiento del sistema.

Ahora bien esta especificación debe de cumplir con ciertas características, debe de ser exacta, consistente, completa y autoritativa. Por exacta queremos decir que no deben existir circunstancias para las cuales no se pueda distinguir un funcionamiento aceptable de uno inaceptable. Un especificación es autoritativa desde el punto de vista que no debe de tener dudas sobre el buen o mal funcionamiento de un sistema y no admite una autoridad mayor. Se considera completa si esta definido para todas las posibles circunstancias. Así mismo debe de ser capaz de tomar una decisión aunque las circunstancias produzcan situaciones conflictivas.

Cuando ocurre una falla en un sistema tres preguntas se plantean enseguida: ¿cómo?, ¿donde? y ¿cúanto? Para poder responder a estas preguntas es necesario que un sistema cuente con modulos específicos para cada una de ellas. Estos módulos son:

1. *Detección de error:* para volver un sistema tolerante a las fallas, hay que, en un principio, definir una estrategia para detectar los errores que han provocado una falla.

2. *Estimación de daños:* una vez que el error es detectado, hay que medir el nivel de daños ocasionados al sistema. Es posible que cierta información maliciosa haya sido difundida.

3. *Restablecimiento de error:* las técnicas consideradas para que un sistema pueda restablecerse tienen por objetivo transformar el estado erróneo del sistema en un estado no-erróneo, gracias al cual el sistema puede recomenzar a funcionar.

4. *Tratamiento de la falla y continuidad del servicio:* es la función que define la activación de los servicios redundantes de manera a asegurar el servicio. Hay que asegurarse que la falla no se reproducira una vez que el sistema esté restablecido, (problema de fallas de modo comun). Detectar un error no es suficiente para poder identificar una falla; ademas hay que asegurarse que el servicio pedido será provisto.

A continuación veremos en más detalle los módulos anteriores

4.1 La detección de error

La mayor parte de las técnicas estan basadas en el concepto de *excepciones*. Se dice una *excepción* es *alcanzada* por un sistema cuando algo fuera de su especificación de funcionamiento ocurre. Una vez que la excepción es alcanzada se invoca a un *manejador de excepciones*, el cual esta definido de acuerdo a la excepción que se alcanzó.

Considere un sistema S, que ha sido diseñado para proporcionar un servicio $f(s)$. Si el comportamiento de S no esta de acorde a la especificado en S entonces S ha fallado.

Ahora bien el objetivo principal de la tolerancia a fallas es prevenir fallas, una de las posibles medidas a tomar para la detección de errores puede ser la intercepción de las salidas producidas por S y verificar si estos resultados estan dentro de lo especificado. Esto trae como consecuencia el diseño de un nuevi sistema S', que consiste de un módulo de servicio orignal y un módulo de chequeo de resultados

El módulo de chequeo debe de basarse solo en la especificación del sistema. No debe haber fallas entre el sistema y el chequeo, (i.e. debe de ser independiente).

En la realidad crear un sistema que cumpla con las especificaciones anteriores es muy difícil. Debido a esto los sistemas solo utilizan chequeos para un nivel de *aceptibilidad.* Esta *aceptibilidad* es un nivel más bajo de lo que sería el perfecto funcionamiento, (i.e. lo que se considera un comportamiento aceptable del sistema), que no ofrece ninguna garantía de ausencia de errores no-detectados.

Existen varios tipos de chequeo, según el tiempo en que se hagan. El más utilizado es el denominado *chequeo de último momento.* En este tipo los chequeos son realizados después de que los resultados han sido generados pero justo antes de que estos resultados sean liberados por el sistema. Dos criterios sirven para sustentar la adopción de este tipo de chequeo:

1. una vez que el resultado dejó el sistema ningún mecanismo de tolerancia a fallas puede aplicarse;
2. se asegura que ninguna actividad del sistema queda sin verificar.

La mayoría de los sistemas utilizan chequeos que pueden caer dentro de la siguiente clasificación:

1. chequeos de replica
2. chequeos de tiempo
3. chequeos de regresión
4. chequeos de codificación
5. chequeos de razonamiento
6. chequeos estructurales
7. chequeos de diagnóstico

4.1.1 Chequeo de replica

Involucra una replica de la actividad del sistema que esta siendo verificada. La replica puede ser vista como una implemantación alternativa del sistema contra la cual la actividad del sistema puede ser verificada para consistencia.

Medidas más poderosas y más caras para detectar errores. El tipo de replica depende del tipo de falla a detectar.

4.1.2 Chequeo de tiempo

Chequei utilizado en el caso de que la especificación de un componente involucre limitantes de tiempo en la provisión del servicio. Un chequeo de tiempo puede proporcionarse en el sistema para determinar si la operación del componente cumple con estas limitantes. Si las limitantes no son alcanzadas entonces el chequeo de tiempo puede alcanzae una excepción de *timeout* para indicar la falla del componente.

4.1.3 Chequeo de regresión

En sistemas donde la relación de los datos de entrada y los de salida es de uno a uno, una alternativa de chequeo puede ser el usar *chequeos de regresión.* Un chequeo de reversión toma la salida de un sistema y calcula lo que debería de ser la entrada. Las entradas calculadas serían comparadas con las entradas actuales y en caso de diferencia se alcanzaría una excepción.

4.1.4 Chequeo de codificación

Se basan en la redundancia dentro de la representación de un objeto, (o conjunto de objetos), que se esta usando en el sistema. Dentro de un objeto se introducen datos redundantes los cuales mantienen una relación fija con los datos, (no-redundantes), que representan el valor del objeto. De esta forma los errores que resulten de una corrupción de cualquier tipo de dato pueden ser detectados.

Un ejemplo de este tipo de chequeo son los bits de paridad. En este caso algunos bits redundantes se asocian con un conjunto de bits dependiendo del número de bits a enviar. Usando estos bits es posible detectar la alteración en uno de en una transmisión de bits de un sitio a otro.

4.1.5 Chequeos de razonamiento

Otras medidas que pueden ser adoptadas en un sistema estan basadas en un conocimiento del diseño interno y de la construcción del sistema. Estos chequeos verifican si el estado de los diferentes objetos en el sistema es "razonable", basado en el supuesto uso y proposito de esos objetos.

Un chequeo usado frecuentemente consiste en verificar el rango que puede determinar si el valor de un objeto se encuentra dentro de un rango permitido. Por ejemplo: si se tiene un objeto que almacena el valor de un ángulo, este debe de estar entre los valores 0 y 360.

4.1.6 Chequeos estructurales

Existen dos tipos de chequeos que pueden ser aplicados a estructuras de datos en un sistema computacional. Chequeo de la integridad semántica de los datos, el cual esta relacionado con la consistencia de la información contenida en la estructura de datos. Chequeo de la integridad de la estructura esta involucrado con si la estructura misma es consistente.

Estos chequeos estan relacionados son usados en estructuras de datos complejas que consisten de un conjunto de elementos ligados entre sí mendiante apuntadores, (i.e. listas, colas y árboles). En este caso la redundancia puede presentarse de tres principales formas: contadores del número de elementos, apuntadores redundantes pueden incorporarse a en una estructura y los elementos en una estructura pueden contener información acerca del status.

4.1.7 Chequeos de diagnósticos

Los chequeos de diagnóstico difieren de los medidas discutidas anteriormente en el hecho de que estan involucrados específicamente en la verificación de los componentes que componen el sistema, más que en la verificación del comportamiento del sistema mismo. Un chequeo diagnostico involucra probar un componente con un conjunto de entradas cuyas ssalidas son conocidas. Durante el intervalo que se da entre las aplicaciones de un chequeo de diagnóstico se asume que el componente trabaja satisfactoriamente.

4.2 La estimación de daños

Las medidas y mecanismos de detección de error pueden identificar un error, pero no existe una garantía de que todas las consecuencias del error detectado, (i.e. *daño*), serán identificadas. Por lo que antes de que una metodología de recuperación de error empieze a ejecutarse, es necesario que el sistema adopte estrategias de *estimación de daños* con el objeto de tratar de establecer más precisamente la extensión del daño del sistema.

Las estratégias para la estimación de daños se basan en la estructura sobre la cual el diseñador asume que estará funcionando el sistema. Ya que esta estructura es la que le que permite al diseñador identificar la posible actividad que puede seguir el sistema después de una transición de error, y los posibles flujos de información. La estructura se derivará de las restricciones conocidas, (o asumidas), que seran "depositadas"en el flujo de información del sistema. La aplicación de estas restricciones en un sistema operacional *confina* el daño causado por una falla. Este confinamiento es una consideración importante en la implementación de un módulo de estimación de daños.

La más simple, pero a su vez más pesimista, técnica para una estimación de daños es asumir que una vez que se ha alcanzado una excepción, todos los objetos involucrados, desde el inicio del cálculo hasta que la excepción fue alcanzada, han sido dañados.

Tal estimación de daños es considerada como *estática*, en el sentido que es especificado en el tiempo de diseño del sistema.

Técnicas menos pesimistas para estimación de daños intentan proporcionar medidas *dinámicas* dentro del sistema operacional para refinar un estimado del posible daño. Esto involucra medidas que exploren en cierta forma las estructuras de datos del sistema. Ya que el proposito de esta exploración es detectar errores, las medidas aplicarán chequeos de la misma forma como se describió en la sección anterior.

Las medidas de exploración, cuyo objetivo es el de investigar el tamaño del daño, pueden ser vistas como la busqueda e identificación de acciones atómicas.

Estas médidas de exploración son difíciles de implementar de una forma efectiva en un sistema computacional y normalmente solo son usadas en situaciones muy específicas y simples. En la práctica las situaciones inesperadas y complejas que pueden ocurrir usualmente requieren de alguna forma de asistencia manual para determinar el nivel de daño que fue causado por una falla.

4.3 La recuperación

Una vez que el sistema ha detectado un error, y que se ha determinado el nivel de daño que provoco dicho error, se debe aplicar una técnica de recuperación para continuar operando.

Como se explicó anteriormente una vez que una excepción es alcanzada, se invoca a un manejador de excepciones. Ahora bien para una excepción en particular el diseñador del sistema puede predecir los posibles errores que pudieron provocar dicha excepción. En este caso estamos hablando de un *daño anticipado*. Por otro lado si un daño no se puede predecir entonces se habla de *daño no-anticipado*.

La importancia de esta diferencia es que para el daño anticipado es posible implementar una recuperación de error eficiente, realizando ajustes muy cuidadosos al estado del sistema.

Por otro lado la única forma de recuperarse de un daño anticipado es adoptar una política de supresión de errores, no cual consiste, normalmente, en reemplazar el estado total del sistema. Esto es usualmente conocido como *reseteo*[2] del sistema. La estrategía de "reseteo"básica es, una vez que es solicitada, dejar al sistema en un estado predefinido anteriormente, por ejemplo un estado inicial del sistema.

A partir de lo anterior podemos decir que en general existen dos tipos de técnicas de recuperación:

1. **Recuperación hacia delante:** El objetivo principal es poner el sistema en un estado de coherencia futura, el cual no ha sido visitado jamás, (por ejemplo el mecanismo de excepción). Esta técnica es eficaz en el caso donde los algoritmos funcionan en ambientes que preceden situaciones previsibles.

2. **Recuperación hacia atrás:** Tiene por objetivo hacer regresar el sistema en un estado anterior a la aparición de la falla. Proporciona una estrategia general muy conveniente para restablecerse de situaciones que un diseñador no había tomado en cuenta.

4.4 La continuidad del servicio

El objetivo de este modulo es el de describir las técnicas para erradicar las fallas del sistema de tal forma que el sistema puede seguir funcionando. Estas técnicas deben proporcionar un tratamiento para la falla misma, (más que para los errores), y pueden dividirse en dos etapas llamadas *ubicación de la falla* y *reparación del sistema*.

Una de las posibles, (y más fáciles), estrategias que se pueden adoptar es la de ignorar la falla y esperar que todo pase bien. Existen tres circunstanicas en las cuales esta política tendría éxito:

1. si la recuperación de los errores generados por la falla es lo suficientemente poderosa para adaptarse con futuras manifestaciones de fallas;

2. si por fortuna la futura ejecución del sistema evita la falla;

3. si la falla es transitoria.

[2] Aunque el termino más correcto es el de reinicialización, *resetear* es una palabra muy usada en el área de la computación.

Para poder tratar con una falla primero se tiene que concocer su ubicación. Ya que la excepción es la que indica la presencia de una falla, es importante asegurarse que las excepciones identifican claramente el componente que falló.

Sin embargo la naturaleza misma de una excepción la hace incapaz de calcular un estimado inicial de la ubicación de la falla. Lo que se necesita es un refinamiento de las excepciones. Este refinamiento involucra ciertas mediadas *exploratorias*.

La técnica exploratoria más importante es la del chequo de diagnóstico. Como se explicó anteriormente esta involucra un componente que posee un conjunto de entradas de las cuales se conocen las salidas correctas. Si las salidas producidas por el componente difieren de aquellas esperadas, entonces puede asumirse que una falla esta presente en ese módulo.

Con el objeto de reducir el costo de diseño del chequeo de diagnóstico se puede utilizar un chequeo de replica para hacer el diagnóstico. Los resultados obtenidos en un componente específico de un sistema son comparados con los resultados del componente conrrespondiente en una copia del sistema. Si los resultados difieren se asume que la falla se encuentra en el componente comparado. En caso contrario se deben realizar otras pruebas.

Cuando la falla se ha localizado uno o más componentes del sistema son considerados como averiados. Para prevenir el hecho de que estos componentes alteren el futuro comportamiento del sistema se necesita una especie de *reparación del sistema*. Se asume que esta reparación no involucra realizar reparaciones internas de un componente.

Las técnicas de reparación estan basadas en *reconfigurar* el sistema de tal forma que las características de uso de los componentes sospechosos son modificados.

Las técnicas de reparación se pueden clasificar en:

- *Manuales:* todas las acciones son realizadas por algún agente externo al sistema, (generalmente un humano).
- *Dinámicas:* acciones realizadas por el sistema, pero solo en respuestas a instruccionbes de su medio ambiente.
- *Espontaneas:* todas las acciones son iniciadas y realizadas por el mismo sistema.

Se debe tomar una decisión importante cuando un sistema regresa a una operación normal después de manejar una excepción que fue señalada por un componente. La cuestión es si la actividad interna del componente puede reanudarse después de que la excepción ha sido manejada por el sistema. Si esta no puede reanudarse entonces el señalamiento de una excepción termina la actividad del componente. En [LiSn79] se discute este aspecto, diferenciando un *módulo de reanudación* de un *módulo de terminación*, dentro de una semántica de esquema de manejo de excepciones.

Se considera recomendable que un manejo de excepciones solo este basado en un modelo de terminación, en particular cuando manejadores de excepcicones son usados exclusivamente para la tolerancia de fallas. Existen dos razones principales:

1. Un mecanismo que solo soporta terminación es muy simple de construir.
2. La reanudación es una respuesta inapropiada a una excepción que indica la presencia de una falla. Ninguna información relacionada con la operación interna de un componente, estaría disponible en el sistema.

5 La redundancia dentro de la tolerancia a fallas

Los principales mecanismos de la tolerancia reposan sobre la existencia de grupos de servidores redundantes de los cuales los elementos pueden tener fallas. Para la concepción de este tipo de mecanismos hay que considerar las etapas siguientes:

1. adoptar una política de decisión de permanencia en el grupo de servidores redundantes. Entre las posibles situaciones que puede afectar lo anterior se encuentran:
    - fallas de los miembros;
    - reinsersiones de los componentes reparados;

2. asegurar las transmisiones fiables hacia los grupos de componentes redundantes (difusión de informacion de cada uno de los clientes hacia el grupo de servidores redundantes implementando un servicio). Estas difusiones deben ser realizadas bajo diferentes hipótesis de fallas;
3. buscar un consenso sobre un valor calculado n veces, (voto repartido);
4. asegurar la satisfacción de coacciones temporales, (que pudieran ser severas en ciertas aplicaciones, por ejemplo, de tiempo real). El ordenamiento en tiempo real distribuido las tareas debe asegurar la satisfacción de las restricciones. Un punto esencial para la detección de fallas temporales como para la satisfacción de restricciones a tiempo real es la existencia de una sincronización de relojes disponibles sobre los diferentes sitios;
5. asegurar el control del acceso concurrente a los datos (conservación de la coherencia), por los datos duplicados, en el caso de un algoritmo que utiliza los datos repartidos compartidos sobre diferentes sitios;
6. preveer la falla de servidores, si la programación de la aplicación incluye servidores centralizados;

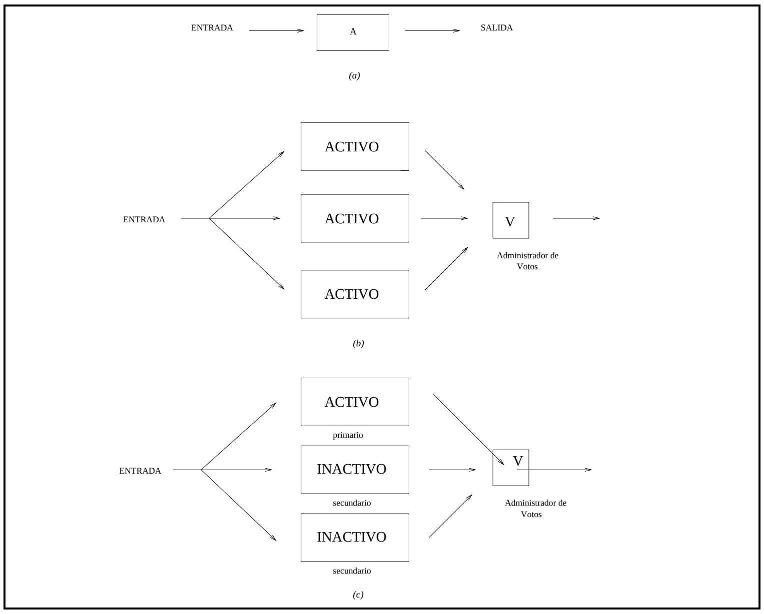

Figura 2: *Redundancia a tres modulos*

Las técnicas para asegurar la tolerancia a las fallas se basan en la noción de *redundancia*. Estas técnicas utilizan elementos suplementarios en el sistema, los cuales son considerados como redundantes en el sentido que no son necesarios en un sistema que no tiene el riesgo de presentar fallas.

Existen viejas arquitecturas que son implementadas en términos materiales o en términos de software pero su organización y su objetivo es exactamente el mismo.

La redundancia puede ser utilizada para simplemente detectar las fallas: el resultado de una incoherencia constatada es simplemente señalar una falla. Puede también ser utilizada para detectar y tolerar las fallas en el cuadro de un procedimiento de reconfiguración que sigue una falla.

Una redundancia puede ser material o de software, cada una teniendo sus propias características y clasificación.

5.0.1 Arquitecturas para la redundancia de software o material

La mayor parte de estas arquitecturas se fundamentan sobre el modulo triple de redundancia, (TMR en inglés). Para volver un módulo A tolerante a las fallas, (parte (a) de la figura 2), A es remplazada por un sistema TMR. Este sistema está compuesto de tres copias del módulo A y de un módulo de voto V, (administrador de votos), que verifica las respuestas de tres módulos y selecciona las de la mayoría. El sistema es, entonces, capaz de tolerar la falla de una de las copias del módulo A, produciendo una respuesta a partir de otros dos módulos, (ver (b) de la figura 2). Encontramos:

I. Redundancia Temporal

Para un componente suceptible de fallas, (intermitentes pero eventualmente temporales o incluso bizantinas), la práctica corriente consiste en intentar disfrazar esta falla por un nombre fijo de tentativas sucesivas.

En general esta técnica es utilizada por los servidores y no es más que después del fracaso de este enfoque que una falla es declarada, (del tipo de aquella para la cual la redundancia temporal ha fracasado).

II. Redundancia Espacial

Un grupo g de servidores redundantes puede ser diseñado de tal manera que a pesar de la falla de ciertos miembros de ese grupo, el servicio que ofrece desde un punto de vista global por los miembros del grupo g sea conservado, con desempeños eventualmente degradados.

El grupo g se comporta como un componente único teniendo su propio nivel de falla superior y el de sus miembros, (las únicas fallas observables por un usuario del grupo g son de un nivel superior; por ejemplo fallas de omisión si el objetivo era disfrazar fallas francas).

III. Redundancia Pasiva

Tiene por objetivo tolerar las fallas francas. En efecto, uno solo de los componentes, (el primario), realiza las operaciones en las que el resultado es afectado a la salida, (ver (c) de la figura 2). En caso de falla del primario uno de los comoponentes inactivos, (secundario) es seleccionado y activado para tomar a cargo el servicio.

A fin que el nivel compuesto primario pueda terminar el servicio, hay que reconstituir un contexto de ejecución correcto que implica:

1. una recopia periódica de información de recuperación constituidos por el componente primario para los secundarios.
2. una reejecución de los servicios provistos desde el último punto de recuperación, en vista de reconstituir la parte faltante del contexto.

IV. Redundancia activa o dinámica

En la redundancia dinámica todos los componentes realizan los tratamientos, (ver (b) de la figura 2).

Los gestores de la redundancia tratan las salidas para tolerar diferentes clases de fallas de servidores.

V. Redundancia selectiva activa

Un solo servidor es afectado a las salidas. En caso de falla del componente primario el secundario toma el control (con el contexto de ejecución completo de la actividad).

El administrador de la redundancia puede ser mas complejo que un simple conmutador. Cuando los dos componentes están activos, puede escojer utilizar los resultados de un componente o del otro según los sitios de residencia.

VI. Redundancias masivas

Se trata de un sistema de seguridad clásico con voto, con o sin diversificaciones de versiones de código.

Si el objetivo es simplemente tolerar las fallas temporales, es posible tomar la primera respuesta disponible. Por otro lado, si el objetivo es más bien tolerar las fallas bizantinas, hay que votar por excluir tanto las salidas no-mayoritarias como las que aparecen demasiado tardíamente.

6 Los algoritmos tolerantes a fallas

En la sección anterior vimos las características de los sistemas tolerantes a fallas, en esta sección veremos las características principales de los algoritmos tolerantes a fallas.

Un algoritmo tolerante a las fallas, como se dijo anteriormente, es aquel que continua trabajando, (tal vez de una manera menos performante), aunque un componente del sistema tenga una falla. Por el contrario un algoritmo que no es tolerante a las fallas cesa de trabajar desde que una falla aparece en el sistema. Un algoritmo tolreante a fallas también es conocido como un algoritmo *robusto* o *confiable*[3].

Encontramos en la literatura varias técnicas y mecanismos para volver un algoritmo distribuido tolerante a las fallas. Seleccionamos cuatro mecanismos representativos, y que engloban todos los principios y propiedades que conciernen la tolerancia a las fallas. La última parte consiste en mostrar la tolerancia a las fallas en los sistemas utilizando RPC. Nuestra selección incluye:

1. *El protocolo a dos fases:* asegura la atomicidad de una transacción haciéndola en dos tiempos:
   (a) en un primer tiempo, guarda la información necesaria para efectuar las operaciones de escritura, en un conjunto denominado *intenciones.* En esta fase se debe de poner atención a no cambiar los datos del sistema. La última acción realizada en esta fase se llama cumplimiento de la transacción;
   (b) en el momento de la segunda fase todas las operaciones de escritura guardadas en el conjunto intenciones son ejecutadas, y todos los datos del sistema son modificados.
2. *Las acciones anidadas:* consiste en considerar los algoritmos como un conjunto de acciones anidadas, (como en un programa Pascal). En este contexto una acción es definida como un cálculo que modifica los datos o una lectura que da el estado de uno o varios objetos, (ver [LaPS81])
3. *El modelo de excepciones:* cuando un procedimiento recibe una demanda para un servicio, y este es atendido se producen respuestas. Podemos dividir estas respuestas en dos grupos: respuestas normales y respuestas anormales. Una respuesta es considerada normal cuando el procedimiento ha realizado, de una manera satisfactoria, el servicio pedido; por el contrario, es anormal en el caso contrario. Las respuestas anormales de un servidor son conocidas como respuestas excepcionales o como excepciones.
4. *El esquema de bloques de recuperación* es una estrategia hecha para que un sistema continue proporcionando servicios aunque un sitio tenga una falla. El esquema está basado sobre el hecho que un servicio debe pasar a través de diferentes módulos, (cada módulo trabaja como un filtro).
5. *Técnicas de recuperación en los RPC*: varias de las técnicas usadas para crear RPC's robustos pueden aplicarse para el diseño de componentes tolerantes a fallas. Muchas de estas técnicas estan relacionadas con el modelo cliente/servidor que se vio anteriormente.

7 El protocolo a dos fases

La ejecución de un algoritmo en un ambiente distribuido debe asegurar la atomicidad sin tomar en cuenta el número de sitios implicados en la transacción. Es decir transacción es cumplida o es interrumpida. Cumplir una transacción implica que todas las actualizaciones de ella, así como todas las acciones y mensajes de cumplimiento, tienen lugar en todos los sitios involucrados en la transacción. Por otro lado, interrumpir una transacción implica que ella no afecta a ningún sitio.

El protocolo a dos fases asegura la atomicidad de una transacción haciéndola en dos tiempos:

1. en un primer tiempo, guarda la infomración necesaria para efectuar las operaciones de escritura, en un conjunto nombrado intenciones. Debemos poner atención a no cambiar los datos del sistema. La última acción realizada en esta fase se llama cumplimiento de la transacción;

[3] Robust o reliable algorithms en inglés.

2. en el momento de la segunda fase todas las operaciones de escritura guardadas en el conjunto intenciones son ejecutadas, y todos los datos del sistema son modificados.

Si una falla se produce entre el cumplimiento, y los cambios de datos, la segunda fase se ejecuta de nuevo. Esta última ejecución debe ser realizada tantas veces sea necesario a fin de asegurar que todos los cambios sean realizados.

Por lo tanto el protocolo a dos fases permite a todos los sitios, que participan en una transacción, llegar a la misma conclusión sobre el resultado de la misma. Un sitio, denominado *coordinador*, se encarga de difundir la decisión final de cumplir o de interrumpir la transacción. Esta decisión es tomada un vez que todos los sitios han enviado su acuerdo.

Los protocolos a dos fases presentan el inconveniente de utilizar mensajes suplementarios para asegurar la consistencia, y su ejecución implica la pérdida de autonomía local para permitir una coordinación global de la transacción.

Entre las diferentes implementaciones existentes, (ver [LaSt79, LSGG+79]), consideraremos:

- un protocolo lineal a dos fases;
- un protocolo centralizado a dos fases;

7.0.2 El protocolo lineal a dos fases

Este protocolo considera que una transacción pasa a traves de diferentes estados. El estado inicial es llamado *desconocido*, (ningún cambio ha tenido lugar, o la transacción es terminada). Después que le hemos pedido prepararse al cumplimiento, una transacción pasa al estado *listo* y esto antes de satisfacer esta demanda.

Cuando el sitio se encuentra en este estado no puede interrumpir o cumplir la tranasacción, debe esperar la decisión del coordinador.

Tenemos dos tipos de organización: una lineal y una en árbol. En esta última se utiliza una difusión de cálculo. Si el estado es valido para todos los miembros del árbol, la transacción es validada por en la segunda fase de la difusión.

En la organización lineal, los sitios involucrados en la transacción son ordenados secuencialmente, y el cumplimiento se hace por el intermediario de mensajes que son pasados de sitio en sitio. Cada mensaje contiene el identificador de la transacción y una lista de sitios con el sitio de origen como encabezado.

Es el sitio de origen el que comienza el cumplimiento. Este sitio posee una lista de todos los sitios implicados en la transacción. Envía un mensaje [*preparar*] al siguiente sitio en la lista. Un sitio que recibe un mensaje [*preparar*] verifica si la transacción está activa. Si no lo está, envía un mensaje [*aborta*] al emisor del mensaje. En el caso contrario, pasa al estado LISTO y envía un mensaje [*preparar*], a su sucesor en la lista.

El último sitio de la lista se vuelve el coordinador. Al momento donde el coordinador recibe el mensaje [*prepara*], todos los otros sitios se encuentran en el estado LISTO. En este momento, el último sitio toma la decisión de cumplir o de interrumpir la transacción. Si está activo, y si no tiene condiciones desfavorables, pasa al estado CUMPLIR. En el estado CUMPLIR, el último sitio envía un mensjae de cumplimiento, [*accomp*] a su predecesor. A cada vez que un sitio recibe un mensaje [*accomp*] pasa al estado CUMPLIR y envía un mensaje accomp a su sucesor así como un mensaje de indicación, [*ack*] a su predecesor.

Cada sitio debe guardar un descriptor de la transacción en su memoria hasta que reciba la indicación de mensaje de cumplimiento que ha enviado. Cuando un sitio recibe un mensaje ack, pasa al estado DESCONOCIDO, destruye el descriptor, y ºbliga"la transacción.

La figura 5.3 muestra un esquema de este protocolo.

7.0.3 El protocolo centralizado a dos fases

En este protocolo, más que enviar los mensajes [*preparar*] y [*accomp*] de sitio en sitio, un solo sitio envia todos los mensajes de preparación y de cumplimiento. El cumplimiento comienza en el sitio de origen que posee una lista de todos los sitios.

El protocolo asigna el rol de coordinador al sitio de origen, y permite que todos los sitios de la transacción se preparen y "cumplan"en paralelo. El coordinador pasa a un estado COLECTA, y envía

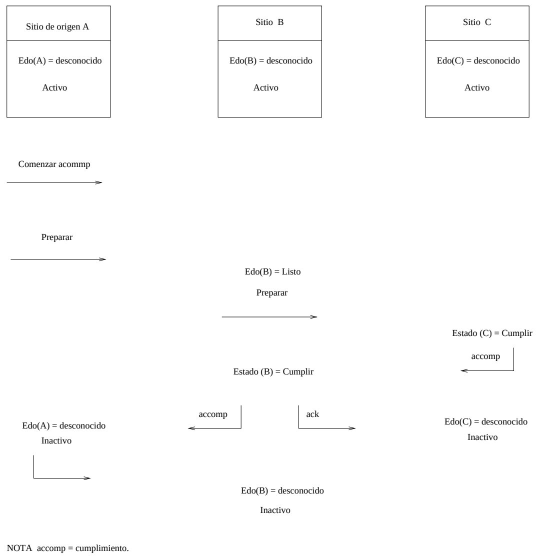

Figura 3: *Esquema del protocolo lineal a dos fases*

un mensaje [*preparar*] a todos los sitios de la transacción. Este mensaje provoca el paso de otros sitios al estado LISTO.

Una vez que un sitio está en el estado LISTO envía una respuesta en forma de mensaje [*vota si*] al coordinador. El sitio espera entonces la decisión del coordinador de cumplir o de interrumpir. El coordinador recibe los votos de todos los sitios. Si un mensaje vota no llega, la transacción se interrumpe. Por el contrario, si todos los sitios responden con [*vota si*], la transacción se cumple. Entonces, si la decisión tomada es de cumplir, el coordinador pasa al estado CUMPLIR y envía un mensaje [*accomp*] a los otros sitios. Cuando un sitio recibe un mensaje accomp, pasa al estado DESCONOCIDO y envía una indicación, (mensaje [*ack*]) al coordinador. Cuando éste recibe el mensaje [*ack*] de todos los sitios, pasa al estado DESCONOCIDO.

La figura 5.4 presenta un esquema de este protocolo.

7.1 Acciones anidadas

Una segunda técnica consiste en considerar los algoritmos como un conjunto de acciones anidadas, (como en un programa Pascal): ver (a) de la figura 5.5) En este contexto una acción es definida como un calculo que modifica los datos o una lectura que da el estado de uno o varios objetos, (ver [LaPS81])

Cada acción está constituida de acciones mas pequeñas, que pueden ser ejecutadas en secuencia así como en paralelo. Cuando una de estas últimas acciones se termina con éxito, sus resultados quedan tentativos a condición que las otras acciones sean también terminadas con éxito.

Si los resultados de una acción atómica cumple sus tentativas; solo pueden ser utlizadas por las acciones atómicas que se encuentran en el sub-árbol de invocación, que tiene por raíz la acción atómica más cercana no-cumplida.

Cada vez que una acción se cumple, guarda su estado. Si al momento de la ejecución, una falla aparece, no tendrá mas que ejecutar la acción anidada implicada.

Las acciones anidadas son una técnica interesante para la concepción de algoritmos distribuidos tolerantes a las fallas, pero presentan el inconveniente de limitar la forma de comunicación entre procesos, así como reducir el grado de concurrencia interno.

Las diferentes implementaciones de las acciones atómicas varían según los argumentos siguientes:

1. la manera de controlar la concurrencia de las acciones;
2. la manera de administrar y de propagar la información que concierne el cumplimiento.

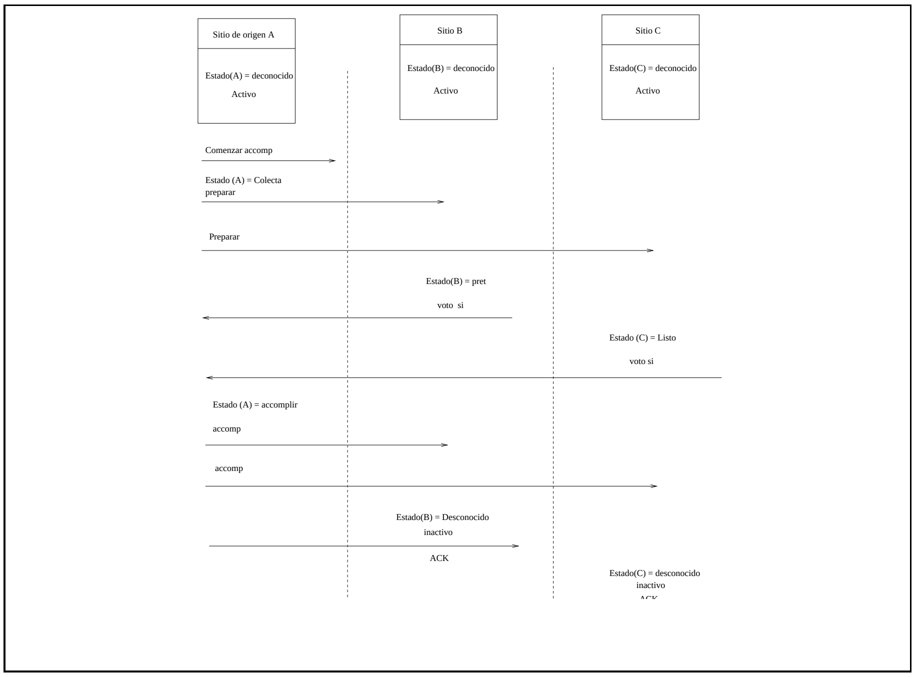

Figura 4: *Esquema Protocolo Centralizado a dos fases*

Varias implementaciones han sido realizadas (ver [Reed83, Moss81, LiSc83, Svo84]). Utilizamos la de Moss para ilustrar las propiedades de las acciones anidadas.

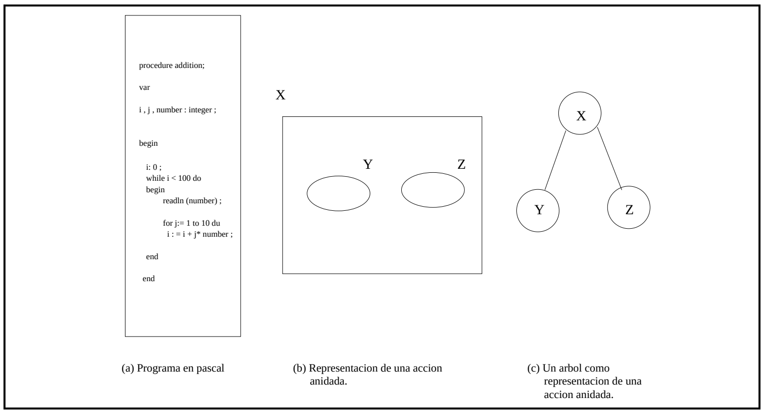

Figura 5: *Las acciones anidadas*

En el modelo de Moss, [Moss81], las acciones son representadas por medio de cuadros. La parte (b) de la figura 5 representa tres acciones: x, y y z. El contorno implica que x posee un rango de acción más grande que y y z, es decir y y z están bajo la acción de x, (otra manera de representarlo es a partir de un árbol: parte c de la figura 5). Los contornos confirman la idea de cada acción es un pequeño universo de sincronización y de tal manera que puede volver a tomar en caso de falla.

La sincronización se construye a partir del concepto de herencia. Supongamos que dos acciones y y z, sean definidas en el interior de una acción x, (ver (b) de la figura 5). Estas dos acciones son ejecutadas y y decide bloquear un objeto.

Si y aún no ha terminado el bloqueo no puede ser liberado, porque si x decide interrumpir el ªborto”, las modificaciones hechas por y no serán tomadas en cuenta. Por lo tanto, en vista de conservar la seriabilidad, hay que asegurarse que las acciones al exterior de x no pueden ver las modificaciones hasta el cumplimiento de x. Una solución consiste en desplazar el bloqueo de y hacia x una vez que y ha terminado, para impedir que una acción al exterior de x pueda tomar el bloqueo del objeto.

Hay dos maneras de "poseerün bloqueo: una acción puede retenerlo o conservarlo. Si un bloqueo es retenido , la acción a un acceso exclusivo sobre el objeto, (no hay más que un solo propietario del bloqueo). Un bloqueo conservado toma el papel de un "guardian", en el sentido donde vigila que las acciones exteriores no puedan tomar el bloqueo, pero permite a las acciones interiores tomarlo. Una vez que la acción es cumplida, sus "padres"guardan todos los bloqueos que poseía la acción cumplida, (se dice que los padres han heredado el bloqueo).

7.2 Las excepciones

Como se mencionó en ??? las excepciones son el métodos utilizado como principal herramienta en la detección de errores. En efecto es gracias a una excepción que podremos darnos cuenta que una falla ocurrió.

Consideremos el modelo cliente/servidor definido en la sección 1.2.3. En este modelo, un procedimiento recibe una demanda para un servicio, y una vez que el servicio es hecho, produce respuestas. Podemos dividir estas respuestas en dos grupos: respuestas normales y respuestas anormales. Una respuesta es considerada normal cuando el procedimiento ha realizado, de una manera satisfactoria, el servicio pedido; por el contrario, es anormal en el caso contrario. Las respuestas anormales de un servidor son conocidas como respuestas excepcionales o como excepciones.

Cada vez que un procedimiento es llamado, decimos que un procedimiento es invocado a partir de una instrucción de invocación que hace parte del procedimiento que llama. La invocación provoca una invocación del procedimiento invocado, y llamamos punto de invocación al lugar a partir del cual una operación es invocada. En el cuadro de una respuesta excepcional se dice que el que llama ha señalado una excepción en el programa, y que el procedimiento que ha producido la excepción ha quitado una excepción. El procedimiento que detecta la condición excepcional es nombrado el señalador. Las acciones y/o operaciones a realizar para tratar la excepción son especificadas en un procedimiento llamado encargado de excepción.

Cuando una ejecución es quitada, el sistema se confronta a dos problemas:

1. ¿cuál procedimiento "jugará"el papel del encargado de excepción?
2. una vez que la excepción sea quitada, ¿es necesario que el señalador continue existiendo?

En el primer problema los candidatos más adecuados son los procedimientos activos en el momento mismo que la señal se produce.

El segundo problema implica una elección entre la complejidad del algoritmo y la simplicidad del código. Dos modelos son factibles: el modelo de recuperación o el modelo de terminación.

7.2.1 El modelo de recuperación

El señalador continua existiendo después de la señalización de la excepción. El encargado de excepción es diseñado de tal manera que la ejecución del señalador puede ser recuperado más tarde. El hecho de que la recuperación afecte la interrelación entre los procedimientos vuelve al modelo más complejo que el de la terminación.

La figura 6 muestra un esquema de este modelo. Consideremos tres procedimientos P, Q y R. P invoca a Q y Q invoca R. Si R señala una excepción r, entonces Q debe ocuparse de ella. Sea Hr el encargado de excepción, definido en Q, que se ocupa de r. Al momento del tratamiento de r, Hr puede señalar una excepción q, ésta debe ser tratada por P, (porque P es el que llama a Q). Se Hq el encargado de excepción que se ocupa de q. Cuando Hq termina, Q es consumado en Hr, R puede continuar su ejecución solamente una vez que Hr ha terminado. Hay que remarcar que la información que concierne las señalaciones circula hacia lo alto, mientras que la información de las "terminaciones"circula hacia lo bajo.

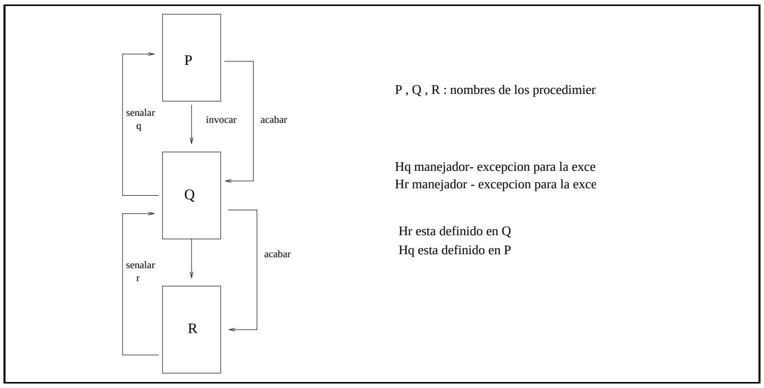

Figura 6: *La circulación del control en un modelo de recuperación*

7.2.2 El modelo de terminación

La aparición de una excepción obliga al señalador a terminar su ejecución. El procedimiento puede terminar de diferentes maneras; puede haber terminado ya sea normalmente o en condiciones excepcionales. Es deseable que el procedimiento informe el sitio que llama de la manera de la cual ha terminado. Para esto, puede utilizar un número definido a partir del número y del tipo de condiciones de terminación posibles. El modelo utiliza un mecanismo linguístico más simple que el del modelo anterior.

7.2.3 Los mecanismos de encargado-excepciones

Los mecanismos de los encargado-excepciones difieren en la manera en la cual llaman las operaciones necesarias a la recuperación. En [Good75] Goodenough da una lista de los principales mecanismos:

1. **Llamado de sub-rutinas:** la primera técnica utiliza el llamado de sub- rutinas predefinidas, utilizando tres tipos de parámetros:
   (a) **sub-rutinas:** el encargado-excepción es diseñado como una sub-rutina y su nombre es utilizado como parámetro de un llamado.
   (b) **etiqueta:** el encargado-excepción comienza su ejecución en el lugar indicado por la etiqueta utilizada como parámetro.
   (c) **variables de estatuto:** se refiere a un parámetro de tipo entero, que indica si una excepción ha tenido lugar y, en caso positivo, cual.
2. **Orientados a objetos:** el encargado-excepción está ligado a un objeto; el control es transferido al encargado-excepción cuando el objeto es utilizado en ciertas operaciones y si ciertas condiciones sean cumplidas.
3. **Llamados de SET-UP:** se refiere a una operación para asociar un encargado-excepciones a una excepción que puede ser retirada (leve) más tarde.
4. **Condiciones PL/I:** utilizan las propiedades de las condiciones ON del lenguaje PL-I. A pesar de la complejidad de las condiciones, éstas permiten asociar una sub-rutina definida por el programador a una condición de excepción.
5. **Técnicas de rango de faltas:** están basadas en la idea de que es imposible tolerar ciertas faltas. Entonces, se trata de escoger una operación alternativa que puede producir el mismo efecto que la operación interrumpida.

7.2.4 Sintáxis de una excepción

La sintáxis de una excepción varía de una implementación a otra, (ver [LiSn79, Flav82, Rob89]). En la figura 7 presentamos la sintáxis que permite la incorporación de excepciones primitivas, a las unidades sintácticas.

```
<function call>    ::= <function name>  [(<parameter list> ) ]  [<handler group>]
<procedure call>   ::= CALL<procedure name>  [(<parameter list> ) ]  [<handler group>]
<assigment>        ::= <variable> = <expression>  [<handler group>]
<loop>             ::= DO <loop form>; = [<statement>~ END [<handler group>]
<statement-group>  ::= DO; [<statement>~] END; [<handler group>]
<quantiy>          ::= { <constant>  ||
                         <variable>  ||
                         <function call> }
<term>             ::= { < quantity >  ||
                         ( < expression > ) }
<expression>       ::= { < unary operator> <term>  ||
                         <term> < binary operator > <term>  ||
                         ( <expression> ) [<handler group>] }
<subroutine defini ::= <label> PROCEDURE [ ( <parameter name list> ) ]
                         [< exception declaration >]; [<statement>~]
                         END ; [< handler group>]
<handler group>    ::= { { <expression name> : } ~ <statement>~ } ~
```

Figura 7: *Sintáxis general para el manejo de excepciones*

Por otra parte, en la figura 8 presentamos un ejemplo de lo que podría ser un compuesto ideal para el tratamiento de excepciones.

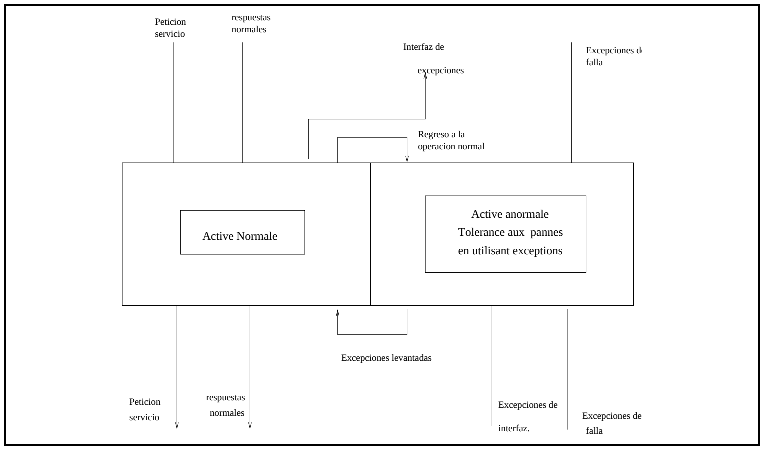

Figura 8: *El componente ideal para tratar excepciones*

Para ilustrar las propiedades de las excepciones en un ambiente más práctico mostramos el mecanismo de excepciones utilizado por el lenguaje ADA.

7.2.5 Las excepciones en ADA

Las excepciones en ADA utilizan la sintáxis siguiente para señalar y quitar (lever) una excepción:

```
begin
    <BLOQUE DE INSTRUCCIONES>
    exception
        when <id_exception> =>
            <ENCARGADO-EXCEPCION>
end
```

Si la excepción identificada por id-exception es quitada (leve) en el momento de la ejecucipon del bloque instrucciones, el flujo de control es interrumpido y transferido a las instrucciones que siguen después del operador $\Rightarrow$.

Los encargado-excepción asociados a cada excepción están definidos a partir de la instrucción **when**. Es posible describir varios encargado-excepciones entre las instrucciones *exception* y *end*.

Por ejemplo consideremos el procedimiento :

```
procedure A (<parametros>)
            :
    calculo(¡parametros>)
            :
    exception
        when error_parametros =>
            PUT ("Error en los parametros");
        when error_numerico =>
            PUT ("Numero muy grande");
        when memoria_insuficiente =>
            PUT ("la operacion pedida requiere de mas memoria")
end A
```

El procedimiento A llama al procedimiento *calculo*. Al momento de la ejecución del cálculo, tres errores pueden presentarse. Una excepción es asociada a cada error posible y un mensaje es emitido en función de la excepción. Notemos la similitud con la instrucción *case* del lenguaje Pascal.

Las excepciones pueden aparecer en cualquier sitio, y tienen acceso a todas las variables declaradas en el programa. El control no puede jamás ser devuelto al programa donde la excepción ha sido quitada. El bloque de instrucciones que sigue al operador $\Rightarrow$ reemplaza el resto del programa. Este bloque de instrucciones contiene el encargado-excepción y acaba así la ejecución del programa.

Si un cuadro no proporciona el encargado-excepción para un caso dado, la excepción es propagada dinámicamente, es decir que el cuadro es abandonado y la excepción es quitada (leve) al punto donde el cuadro ha sido lanzado.

ADA proporciona cinco excepciones predefinidas por la detección de situaciones inhabituales pero previstas. Sin embargo, estas excepciones no serán suficientes en el caso de un error no previsto. La declaración de una excepción está hecha de una manera semejante a la de una varibale, y para quitar una excepción, hay que utilizar una instrucción raise explícita mencionando la excepción:

raise $<$ id_exception $>$

La excepción será propagada al cuadro conteniendo el bloque donde ha sido quitada, si por ejemplo un procedimiento gestion llama al procedimiento siguiente:

```
procedure busqueda (llave)
            :
    exception
        when error_parametros =>
            PUT("No se encontro el elemento");
                raise no-encontrado
        when others
            :
end busqueda;
```

La excepción *no_encontrado* será señalada en el procedimiento *gestion*, y un mensaje será emitido.

Algunas veces es practico tratar la excepción después de propagarla, lo que puede realizarse usando la instrucción:

raise

por ejemplo podemos escribir:

```
begin
        :
    when others ⇒
        PUT(<meensaje de error>);
        HACER_LIMPIEZA
        raise
end
```

7.3 El esquema de bloques de recuperación

Fue introducido por la primera vez en [HLMR74], en donde encontramos varias implementaciones, (ver [Wen+78, Shri78, Lee+80]). El esquema de bloque de recuperación es una estrategia hecha para que un sistema continue proporcionando servicios aunque un sitio tenga una falla. El esquema está basado en el hecho que un servicio debe pasar a través de diferentes módulos, (cada módulo trabaja como un filtro).

El primer módulo es llamado *módulo primario.* El se ocupa de realizar todas las operaciones necesarias para *cumplir* con el servicio solicitado. Se supone que este módulo ha sido suficientemente probado y revisado; sin embargo, es posible que contenga ciertas fallas en el diseño, que puedan provodar una falla del sistema.

El segundo módulo es el módulo de prueba de aceptación, está compuesto de las instrucciones necesarias para verificar que el módulo primario haya terminado correctamente. Es ejecutado fuera de módulo primario.

Si un error se produce en este módulo primario un módulo alternativo es notificado y llamado para realizar el servicio pedido o para encontrar alguna otra solución. La prueba de aceptación es hecha de nuevo. Un bloque de recuperación puede contar varios módulos alternativos. Los bloques de recuperación pueden estar anidados, de tal modo que un bloque pueda hacer parte de un otro bloque. Cada uno de estos módulos presenta que se acabamos de describir.

Un bloque de recuperación debe contener los elementos siguientes:

- un módulo primario;
- una prueva de aceptación;
- un módulo alternativo.

La figura 9 muestra como son arreglados estos elementos diferentes. Al momento de su ejecución, un bloque realiza las siguientes operaciones

1. Ejecución del módulo primario;
2. Una vez que el módulo primario ha terminado su ejecución, la prueba de aceptación está hecha.
   - si el resultado de la prueba no concierne ninguna excepción entonces:
     (a) se asume que los resultados del modelo son válidos;
     (b) se deja el bloque.
   - si la prueba encuentra un error entonces:
     (a) la secuencia de ejecución mostrada anteriormente es repetida, pero en lugar de trabajar sobre el módulo primario, trabajamos sobre uno de los módulos alternativos.
     (b) si todos los módulos tienen una falla entonces:
       – ésta es considerada como una falla de bloque de recuperación.
       – una excepción apropiada es señalada.

```
Bloque A
    ensure  < pueba de aceptacion >
    by      < modulo primario>
    else by < modulo alternativo 1 >
    else by < modulo alternativo 2 >
                :
                :
                :
    else by < modulo alternativo n >
    else error
end bloc
```

Figura 9: *Sintáxis de un bloque de recuperación*

7.4 La tolerancia a las fallas en los sistemas que utilizan APD

Una de las t'ecnicas más utilizadas en algoritmos distribuiods son los RPC. Existe diversos documentos de .... Comenzamos por mostrar el comportamiento de un sistema que utiliza APD para cada tipo de falla.

I. falla franca

El APD respeta el modelo cliente/servidor, (ver sección 1.6). En la sección 5.3.1 ha sido mostrado que los efectos de la falla del cliente sobre un sistema son diferentes de aquellas provocadas por la falla del servidor.

falla del servidor

La consecuencia más importante de una falla del servidor es que la respuesta no regresa jamás al cliente. El cliente puede actuar de dos maneras:

1. puede esperar indefinidamente la respuesta que no vendrá jamás; obligando finalmente a abandonar el programa;
2. puede utilizar un *tiempo de espera* por el sitio que llama; en este caso el servidor debe señalar la ausencia de respuesta al cliente y efectuar las acciones siguientes:
   (a) una recuperación hacia atrás (si es posible),
   (b) una nueva tentativa de ejecución bajo un otro sitio,
   (c) una recuperación sobre el mismo sitio después tolerancia; lo que puede provocar varias ejecuciones del mismo requerimiento, (idempotencia).

falla del cliente

La falla del cliente vuelve al servidor huérfano, (ver sección 5.5.5.a). Una vez que el cliente está restablecido debe ser capaz de hacer la diferencia entre las nuevas respuestas y las de los requerimientos viejos. Por lo tanto hay que destruir el proceso servidor y distinguir los requerimientos.

II. falla de omisión (de mensajes)

Hay APD implementados, para razones de eficacia, encima de una comunicación no fiable, (por ejemplo el sistema CHORUS). Estos APD deben contar con un mecanismo de tratamiento de pérdidas de mensajes. Este mecanismo puede ser realizado de diferentes maneras: que la respuesta satisfaga la demanda, que el próximo requerimiento satisfaga la respuesta, o que el cliente efectúe varias ejecuciones del mismo requerimiento.

III. Fallas Temporales (del servidor)

La respuesta llega después de una fecha de caducidad definida por un tiempo de espera del cliente, (caso de los sistemas industriales). En este caso le corresponde al cliente escoger las acciones en función de la aplicación.

IV. Fallas Bizantinas (del servidor)

Este tipo de fallas exige un tratamiento excepcional. Una solución posible es utilizar una redundancia masiva, (varias ejecuciones en paralelo del mismo requerimiento), con un algoritmo de consenso sobre las respuestas del tipo voto bizantino.

7.4.1 Tratamiento de los huérfanos

Los huérfanos son las ejecuciones de servidor que continuan existiendo cuando el cliente tiene una falla. En la figura 5.10, presentamos un ejemplo: el procedimiento A ejecuta un APD sobre B, este procedimiento envía un APD sobre los sitios C, D y E. Después de cierto tiempo, el sitio B tiene una falla, se dice entonces que los sitios C, D y E son huérfanos del sitio B.

La aparición de los huérfanos puede provocar la realización de trabajos inútiles, así como el derroche de recursos que ya no son accesibles por las tareas útiles.

Devemos hacer un análisis de las modificaciones del estado del cliente y del servidor, porque sus estados pueden volverse inconsistentes. También hay que poner en lugar de las estrategias de abandono de tratamiento de los servidores en caso de falla del cliente, y asegurarse que los servidores dejen un estado coherente de las ejecuciones. varias soluciones son factibles:

I. Exterminación de un huérfano

Si el cliente se da cuenta que el servidor tiene una falla, entonces escoge otro servidor para recomenzar.

En este fin el cliente nota en memoria estable todos los APD en curso. Cuando un sitio cliente es reactivado, examina el historial de los APD no terminados y pide la destrucción de todos los servidores huérfanos otra vez activos.

Las principales dificultades que presenta esta técnica son:

- la busqueda de los huérfanos puede ser larga y difícil en el caso de llamadas en cascada;
- la destrucción de los huérfanos es condicionada por la reactivación del cliente.

II. Reencarnación de un huérfano

Algún otro sitio toma el relevo del sitio que tiene una falla. En este caso, todas las operaciones necesarias al relevo son transparentes para el cliente.

El cliente introduce en cada uno de sus requerimientos un número de época correspondiente a sus periodos de actividad sucesivos (registrado en memoria estable).

Cuando un cliente es reactivado, cambia de época. Sea que el advierte todos sus sitios servidores de cambio de época para que destruyan los llamados todavía en ejecución; o sea que le haga llamados poco a poco.

III. Expiración (vencimiento)

La ejecución de un servidor está siempre asociada a una vigilancia periódica del cliente: el servidor arma las demoras de guardia.

Si la ejecución se termina antes del vencimiento, el servidor transmite su respuesta recuperando o no recuperando su cliente.

Si el quantum de tiempo se acaba, el servidor pregunta a su cliente si él está todavía operacional, entonces:

- si el cliente responde, se arma un nuevo retraso y se continua.
- si el cliente tiene falla, se hace un abandono coherente de ejecución.

7.4.2 APD resistentes a las fallas

Varios tipos de APD tolerantes a las fallas han sido implementados, utilizando técnicas diferentes para cada uno. En esta sección, discutimos las propiedades de cuatro APD representativos de las técnicas mostradas anteriormente.

En [JaGa85] encontramos un APD atómico. La concurrencia está controlada incorporando un identificador de camino de llamado en cada mensage que representa un APD. Cuando un sitio tiene una falla sigue la política de exterminación de los huérfanos. Cada procedimiento conserva en sus tablas los APD que ha recibido o que ha hecho. Estas tablas son utilizadas para anular los efectos provocados por los APD lanzados.

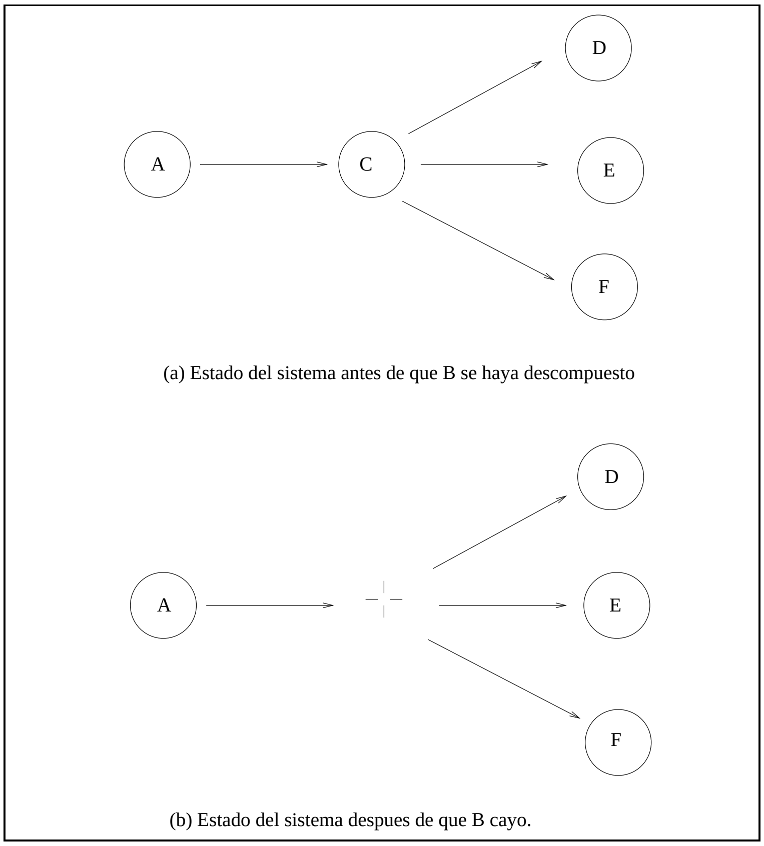

Figura 10: *Ejemplo de sitios huerfanos*

Es en [RaCh89] que encontramos el término de huérfanos. Distingue dos tipos de APD con relación a sus niveles de idempotencia:

1. **APD orientado:** el servidor debe guardar la relación de orden de las llamdas que ha realizado.
2. **APD inferior:** el servidor no necesita guardar una relación de orden.

En cuanto el servidor lanza un APD, posee un punto de recuperación dependiendo del tipo de ASP. Si un sitio tiene una falla que provoca la aparición de huérfanos, algún otro sitio podría adoptarlos tomando en cuenta los puntos de recuperación poseidos anteriormente por el servidor y los huérfanos. Esta ªdopción"sería, en el caso ideal, completamente escondida al usuario.

El APD descrito en [ShPa82] es un ejemplo de exterminación. Utiliza dos parámetros en cada APD lanzado: un retraso de guardia y una variable de statuto en cada mensaje enviado por el servidor. El retraso comprende el tiempo que el cliente debe esperar para la respuesta, y la variable de statuto indica si el servidor a podido hacer el servicio, o si tiene una falla. Se asigna un número secuencial único, (NS) a cada mensaje, si una falla tiene lugar el servidor debe guardar el NS mas grande. Entonces, una vez el sitio restablecido, será capaz de identificar los mensajes "bastardos"que llegan, producidos antes y durante la falla.

La solución propuesta en [YaJT88] se apoya sobre la existencia de un conjunto de sitios candidatos que pueden remplazar un sitio que tiene una falla, (este concepto es utilizado en ARGUS bajo el nombre de guardianes, ver sección 1.5.6). Varias copias del procedimiento son disponibles, (ellas son llamadas encarnaciones). Si el llamado original se suspende, una de las copias toma el papel del sitio que tiene la falla y proporciona el servicio pedido. Cuando una acción es hecha, todas las encarnaciones son ejecutadas concurrentemente. Hay dos tipos de encarnaciones: primarias y secundarias. Estas últimas son ordenadas según su facultad a tomar el papel primario. La figura 11 muestra un esquema representando este modelo de APD.

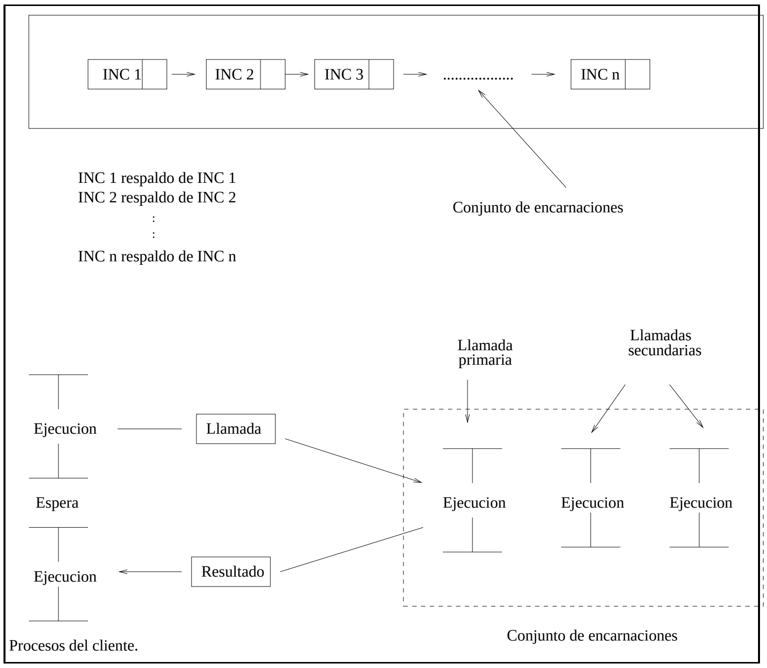

Figura 11: *Esquema de encarnación*

8 El problema de los Generales Bizantinos

Es un problema que ejemplifica la coherencia de los datos dentro de un sistema distribuido. El planteamiento es el siguiente:

Una ciudad se encuentra citiada por varios ejercitos bizantinos cada uno con un general al frente de ellos. Se trata de que los generales decidan, de acuerdo a sus opiniones, si se retiran o si atacan, y qye dicha decisión sea "buena"y ünanime". proviene del hecho de que los generales solo se comunican de dos a dos, y a través de un mensajero. El principal problema esta en que entre los generales pueden existir traidorea, que intentan impedir que los generales leales lleguen a un acuerdo. Los generales deben contar con un algoritmo que garantiza que:

1. Todos los generales leales decidan el mismo plan de acción
2. Un número pequeño de traidores no pueda causar que los generales leales adopten un mal plan.

Resumiendo, un general de comando debe enviar una orden a sus generales tenientes de tal forma que:

1. todos los tenientes obedezcan la misma orden;
2. si el comandante general es leal, entonces cada teniente leal obedezca la orden que el envio.

Estas dos condiciones se conocen como condiciones de *consistencia interactiva.* Hay que notar que si el comandante es leal, entonces la condición 1 sigue de la condición 2. Sin embargo el comandante puede ser no leal.

Se puede demuestrar que no existe una solución para tres generales dentro de los cuales exista un traidor. La solución que se propone funciona para el caso de $3m+1$ o más generales en presencia de a los más m traidores.

Se asume que cada general ejecuta un algoritmo que involucra el envíio de mensajes a otros generalesi, y también se asume que un general leal ejecuta correctamente este algoritmo. Un comandante traidor puede decidir no enviar ninguna orden. Ya que los tenientes deben ejecutar alguna orden, necesitan una orden a obedecer por default. En este caso dicha orden es RETIRADA.

La solución se plantea en terminos de El algoritmo se muestra a continuación:

Algoritmo OM(0)
El comandante envía su valor a cada teniente
Cada teniente usa el valor que recibe de su comandante, o usa el valor RETIRADA si no recibe ningún valor.
Algoritmo OM(0), $m > 0$
El comandante envia su valor a cada teniente.
Para cda i, v_i = valor teniente i recibio del comandante

Se asume lo siguiente:

1. Cada mensaje que es enviadop es recibido correctamente.
2. El receptor de cada mensaje sabe quien lo envío.
3. Se puede detectar la ausencia de un mensaje.

9 Conclusiones

Hemos mostrado diferentes técnicas y mecanismos utilizados en la tolerancia a fallas. Aunque no existe una técnica completamente segura para crear un sistema tolerante a fallas, se pueden mezclar diferentes técnicas para diseñar un sistema que tolere fallas.

Cinco técnicas han sido presentadas, hemos comenzado por el protocolo a dos fases, como lo hemos explicado, se trata de un protocolo para asegurar la atomicidad de una transacción. Esta técnica es muy apropiada a las operaciones realizadas en las bases de datos o en los algoritmos de terminación.

Las acciones anidadas presentan una buena solución para los algoritmos que utilizan procedimientos ªnidados".

Los bloques de recuperación presentan propiedades convenientes a ambientes que cuentan con una memoria común. Presentan un esquema muy particular y utilizan modelos alternativos para el tratamiento de una falla.

Las técnicas utilizadas para volver los APD resistentes a las fallas son específicas a los APD. Aunque el APD sea un útil de la ola recursiva, no es suficiente volver el APD tolerante a las fallas. En efecto, el objetivo de todas estas técnicas es de volver los APD robustos, independientemente de las características del ambiente en el que sean utilizados. Nuestro modo es intentar volver resistente la ola recursiva, y no únicamente cada uno de los APD que la componen.

Es importante remarcar un hecho, no se puede crear un sistema que soporte todo tipo de fallas. La práctica más común es seleccionar un determinado tipo de falla que sea suceptible de aparecer de acuerdo a la naturaleza del algoritmo y a la plataforma sobre la cual se implementó dicho algoritmo. A partir de de esta selección se elebora un algoritmo tolerante a ese tipo de fallas.

Otro detalle a mencionar es sobre la complejidad de los algorimos tolerantes a falla. El transformar un algoritmo en robusto tiene un costo que afecta de forma directa a la complejidad. Un algoritmo robusto siempre tendra una complejidad más grande que uno que no lo es.

Referencias

[AnLe81] Anderson T, Lee P. - *Fault tolerance, Principles and Practice* Prentice Hall International, 1981

[BeGo83] Berstein P, Goodman N. - *Concurrency Control and Recovery for Replicated Distributed Databases* Research Rapport No. TR-20-83, Computer Science Department, Hardvard University.

[BeHG87] Bernstein P, Hadzilacos V, Goodman N. - *Concurrency Control and Recovery in Database Systems* Addison Wesley 1987

[Desw90] Deswarte Yves - *Tolérance aux fautes, sécurite et protection dans les systèmes répartis* Rapport Recherche No. 90.217, LAAS Juillet 1990, Frace

[Flav82] Flaviu Cristian - *Exception Handling and Software Fault Tolerance* IEEE Trans. on Computers, Vol. C-31, No. 6, June 1982, pp. 531-540

[Flav83] Flaviu Cristian - *On Exceptions, Failures and Errors* Research Report No. RJ4130 IBM Research Laboratory, San Jose California, 12/9/83

[HLMR74] Horning J.J, Lauer H. C, Melliar-Smith P.M. and Randel B. - *A program structure for error detection and recovery* Lecture Notes in Computer Science, 16, April 1974, pp. 177-193

[JaGa85] Jay-Kwei, Gannon J.D. - *Atomic Procedure Call* IEEE Trans. on Soft. Eng. Vol. SE-11 No. 10 October 1985

[Lap88] Laprie Jean-Claude - *Surete de fonctionnement et tolérance aux fautes: concepts de base.* Rapport Recherche LAAS No. 88.287 Novembre 1988, France

[LaSt79] Lampson B.W, Sturgis H.E. - *Crash recovery in a distributed data storage system.* Rapport de Recherche No. 69 XEROX, Palo Alto Research Center April 26, 1979

[LaPS81] Lampson B.W, Paul M. and Siegert H.J. - *Atomic Transactions.* Distributed Systems Arch. & Implementations: An advanced Cours Lecture Notes in Computer Science 105, Chapter 11 Edts. New York: Springer-Verlag 1981

[Lee+80] Lee P.A, Ghani N, Heron K. - *A recovery Cache for the PDP-11* IEEE Transactions on Computers, Vol. C-29, No. 6, June 1980, pp. 546-549

[LiSc83] Liskov B, Robert Scheifler R. - *Guardians and Actions: Linguistic Support for Robust, Distributed Programs* ACM Trans. on Prog. Languages and Systems, Vol 5, No. 3 July 1983 pp. 381-404

[LiSn79] Liskov B, Alan Snyder - *Exception Handling in CLU* IEEE Trans. on Soft. Eng. Vol. SE-5 No. 6 November 1979

[LSGG+79] Lindsay B, Selinger P, Galteri C, Gray J.N, Lorie R, Price T, Putzolu F, Wade B. *Transaction Initiation, Migration and Termination* Chapter V of Notes in Distributed Systems Research Report No. RJ2571 IBM Res. Lab., San Jose California, 1979

[Moss81] Moss Eliot J. - *Nested Transactions: An Approach to Reliable Distributed Computing* Technical Report No. MIT/LCS/TR-260 Lab. Comput. Sci. Massachusets Inst. Technol., Cambrigde, MA April 1981

[RaCh89] Ravindran K, Chanson S.T. - *Failure Transparency in Remote Procedure Calls* IEEE Trans. on Computers Vol. 38 No. 8 August 1989

[Reed83] Reed P. David - *Implementing atomic actions on decentralized data* ACM Trans. Comp. Systems vol. 1, Februry 1983 pp. 3-23

[Rob89] Roberts S. Eric - *Implementing Exceptions in C* Research Repport No. 40, DIGITAL Systems Research Center, Palo Alto California, March 21, 1989

[ShPa82] Shrivastava S.K, Panzieri F. - *The design of a Reliable Remote Procedure Call Mechanism* IEEE Trans. on Computers Vol. C-31 No. 7 July 1982

[Shri78] Shrivastava S.K.- *Sequential Pascal with Recovery Blocks* Software - Practice and Experience, Vol. 8, No. 2, March 1978, pp. 177-185

[Stro84] Strong Ray - *Problems in fault tolerant distributed systems* Research Report No. RJ4519 IBM Res. Lab. San Jose Cal. Nov. 79

[Svo84] Svobodova Liba - *Resilient Distributed Computing* IEEE Trans. on Soft. Eng. Vol. SE-10 No. 3 May 1984

[YaJT88] Yap K, Jalote P, Tripathi S. - *Fault Tolerant RPC* Proceedings of 8th. International Conference on Distributed Computing Systems, San Jose California CA, USA, 13 june 1988, pp. 48-54

[Wen+78] Wensley, Lamport, Goldberg, Green, Levitt, Smith, Shostak, Weinstock - *SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control* Proceedings of the IEEE, Vol. 66, No. 10 October 1978, pp. 1240-1255