Tesis de Licenciatura

Un marco integral para el diseño y construcción de sistemas usando una arquitectura basada en colas

Director: Gustavo Pifarré

Tesista: Graciela Morena

1998

Tabla de Ilustraciones

Resumen

La Arquitectura de Software ha atraído un gran interés en los últimos años. Hay varias propuestas para definir un marco integral para la construcción de arquitecturas de software. Se intenta encontrar un modelo para definir y validar una descripción de alto nivel de un sistema. Hay un interés creciente en la industria para proveer técnicas y herramientas que permitan la interoperabilidad entre diferentes tipos de componentes y provean la necesaria flexibilidad para expresar características del sistema. Las herramientas son llamadas lenguajes de interface de módulos o "middleware" y se desarrollan en función de un modelo conceptual. Uno de los modelos más recientemente usado en la industria es el de mensajería y colas. En este proyecto se provee un marco para la construcción de arquitecturas de sistemas de alto nivel basadas en el modelo de mensajería y colas. Este marco permitirá el diseño de una arquitectura de alto nivel y la construcción de una estructura del sistema.

Abstract

Software Architecture has attracted a great deal of interest in recent years. There are several proposals for defining an integrated framework for the construction of software architecture. They try to find a model where any high level description of a system can be defined and validated. There is an increasing interest in the industry for providing techniques and tools that allow the interoperability between different types of components and provide the necessary flexibility to express the features of the system. The tools are called module interface languages or ``middleware" and are developed as a function of a model. One of the most recently used models in the industry is the messaging and queuing system. In this project, we intend to provide a framework for building high level software architectures based on a messaging and queuing model. This framework will allow the design of a high level architecture and the construction of the high level structure of the system.

Un marco integral para el diseño y construcción de sistemas usando una arquitectura basada en colas

Introducción

La Arquitectura de Software ha atraído un gran interés en los últimos años. Hay varias propuestas para definir un marco integral para la construcción de arquitecturas de software. En general se intenta encontrar un modelo para definir y validar una descripción de alto nivel de un sistema. Este tema es discutido por los siguientes autores: [Dean y Cordy 012], [Garlan 028, 013] y [Shaw 029], entre muchos otros.

Existe un gran número de razones para la búsqueda de este marco general. En principio, no existe un lenguaje común y completo que permita a los ingenieros de software definir sus sistemas con un gran nivel de detalle. Además, si se pudieran definir reglas en el modelo, se podría pensar en diseñar herramientas que provean cierta validación al diseñador. Otro aspecto importante es la posibilidad de estimar una aproximación de la performance del sistema a través de una modelización analítica o simulada. Por último, el diseñador puede utilizar una librería de modelos de sistema y reutilizarlo cuando sea posible.

La estructura de los sistemas en ambientes distribuidos fuerza a los ingenieros de software hacia técnicas composicionales que ofrecen las mejores posibilidades para la construcción de sistemas escalables y ampliables en forma modular. [Kramer 011] [Garlan 013]

Por otro lado, hay un interés creciente en la industria para proveer técnicas y herramientas que permitan la interoperabilidad entre diferentes tipos de componentes y provean la necesaria flexibilidad para expresar características del sistema [Garlan 028]. Las herramientas son llamadas lenguajes de interface de módulos o "middleware" y se desarrollan en función de un modelo conceptual: comunicación entre procesos, servicios distribuidos, sistemas de colas, etc. Estas herramientas pueden proveer diferentes niveles de robustez, confiabilidad, tolerancia a fallas e interoperabilidad. [Blakeley 030].

Uno de los modelos más recientemente usado en la industria es el de mensajería y colas: los diferentes componentes del sistema interactúan a través del envío de mensajes almacenados en colas. Esta técnica le da al diseñador una herramienta de alto nivel que asegura la comunicación en el sistema conociendo solamente el nombre de la cola en la cual almacenar los datos. En general, este modelo se basa en un modelo de comunicación sin conexión donde no es necesario establecer una sesión con los demás componentes del sistema para permitir comunicación de alto nivel entre ellos.

En este proyecto se pretende proveer un marco para la construcción de arquitecturas de sistemas de alto nivel basadas en el modelo de mensajería y colas. Este marco permitirá el diseño de una arquitectura de alto nivel y la construcción de una estructura del sistema.

Para el proyecto del cual esta tesis formará parte, se analizaron como posibles temas a considerar, los siguientes:

- Un lenguaje para expresar la arquitectura de sistemas basados en un modelo de colas.
- Una herramienta para la especificación de los sistemas que permita especificar el lenguaje anterior.
- Una herramienta ad-hoc para construir una estructura de alto nivel del sistema.
- Un modelo de teoría de colas en el cual pueda mapearse el sistema antes descripto.
- Una herramienta para modelar la arquitectura del sistema utilizando el modelo de colas antes descripto y analizar su correctitud y performance.
- Una herramienta para simular el sistema y analizar su correctitud y performance.
- Una herramienta que permita la construcción de una estructura de alto nivel del sistema usando una herramienta específica de sistema de colas ya existente en el mercado.

Objetivos

Los objetivos del presente trabajo de Tesis de Licenciatura estan compuestos por las siguientes componentes:

- Definición del alcance de las arquitecturas soportadas por el modelo y su especificación.
- Definición de un lenguaje formal que permita representar las arquitecturas incluidas en el modelo descripto.
- Implementación de una herramienta básica que soporte la definición de arquitecturas mediante el lenguaje anterior.
- Diseño de la herramienta para construir automáticamente la estructura de alto nivel que permita la implementación de una arquitectura descripta por el lenguaje.
- Implementación de la herramienta diseñada.

En conclusión, el objetivo de esta tesis es obtener el diseño e implementación de una herramienta para construir automática y dinámicamente un conjunto de procesos ("middleware") capaces de soportar una arquitectura que pueda expresarse a través del lenguaje definido a tal efecto. Las características de tolerancia a fallas, recuperabilidad y alta disponibilidad deberán ser tenidas en cuenta en las distintas etapas del armado de la solución.

1. Definición de las arquitecturas.

En esta sección se definirán y especificarán las arquitecturas soportadas por nuestro modelo. Con este fin, se presentará una descripción de las arquitecturas válidas como entrada para el modelo presentado y características de alto nivel de las mismas en lenguaje natural. Para ordenar esta descripción y facilitar su posterior referencia se diferenciaron operaciones y reglas que cumplirán las arquitecturas válidas.

Se definirá un lenguaje para especificar las arquitecturas soportadas. Este lenguaje se basa en una gramática o expresión regular. Se definirá, también, un lenguaje gráfico sobre el cual podría basarse la implementación de una herramienta gráfica que permita el diseño de las arquitecturas.

El estudio y definición de un lenguaje para especificar las arquitecturas soportadas es punto de partida para el desarrollo de nuestro trabajo. El foco se ubicará en la descripción de las características de las arquitecturas y en la especificación de un lenguaje muy sencillo para representarlas. En el proceso de búsqueda de las propiedades que caracterizan las arquitecturas soportadas, se hizo clara la necesidad de expresar de una manera inequívoca las mismas.

1.1 Entorno donde existen las arquitecturas a representar

En esta sección y previo a la descripción de las características de las arquitecturas soportadas, se presentarán las características generales de los ambientes donde existen estas arquitecturas. Para una descripción más en detalle de los sistemas distribuidos referirse al apéndice A "Características de los sistemas distribuidos".

En la actualidad está en continuo crecimiento el número y magnitud de los sistemas distribuidos implementados. Las arquitecturas soportadas serán sistemas donde existen un conjunto de procesos ejecutando bajo un modelo de procesamiento distribuido en formas cooperativa. El término sistema distribuido, en lo concerniente a las arquitecturas consideradas, se aplica a sistemas con las características como las que siguen [Tel 001], [Mullender 002], [Singhal 003], [Chandy 005] [Tanenbaum 006 007 008]:

- Los procesos corren en varias computadoras que no comparten memoria ni un reloj global. Cada computadora tiene su propia memoria y corre su propio sistema operativo. Comparten un protocolo de comunicación común.

- Las computadoras se comunican entre ellas intercambiando mensajes a través de una red de comunicación.

- Los recursos propios de una computadora y manejados por ésta se dicen locales, mientras que los de otras se dicen remotos. Generalmente acceder a

recursos remotos es más costoso que acceder a recursos locales debido a los retrasos provocados por la red de comunicación y el overhead de CPU causado por procesar los protocolos de comunicación.

- Falta de Conocimiento global: Cada proceso del sistema no tiene un conocimiento global del estado del resto del sistema en forma completa y correcta.

- Falta de Global Naming: Los nombres se utilizan para referenciarse a los objetos del sistema. Los procesos no conocen como referenciar a otros procesos del sistema, ni siquiera pueden conocer su existencia.

- Escalabilidad: El sistema distribuido de procesos que nos interesa puede crecer con el tiempo. El diseño de la herramienta debe favorecer este hecho sin degradar la performance.

- Compatibilidad: se refiere a la noción de interoperabilidad de recursos dentro del sistema. Se podrían nombrar tres niveles de compatibilidad en sistemas distribuidos: binario, ejecución, protocolo.

  En un sistema compatible a nivel binario todos los sistemas ejecutan el mismo conjunto de instrucciones binarias, aunque los procesadores puedan diferir en performance y input-output.

  Compatibilidad a nivel de ejecución existe si un mismo código fuente puede ser recompilado y ejecutado en cualquiera de las máquinas del sistema.

  La compatibilidad a nivel de protocolo es la menos restrictiva. Logra interoperabilidad al requerir que todos los sistemas soporten un conjunto común de protocolos. Así se logra que cada computadora pueda ejecutar su propio sistema operativo sin perder interoperabilidad.

- Sincronización de procesos: Surge la necesidad de sincronizar a los procesos que desean acceder concurrentemente a un recurso común. Hay que prevenirse contra problemas como deadlock, inanición, etc.

- Aunque existen implementados en la actualidad numerosos sistemas operativos distribuidos, el presente Trabajo de Tesis está orientado a aquellos entornos donde no se cuenta con la disponibilidad de alguno de estos. Se pretende suplir las características que poseen los mismos y que fueran necesarias para la solución del problema planteado.

Dentro de sistemas con las características arriba mencionadas existen muchas arquitecturas diferentes de procesos posibles, en la siguiente sección se describirá el subconjunto de arquitecturas con las que trabajaremos.

1.2 Descripción de las características de las arquitecturas soportadas

En esta sección se describen las arquitecturas de procesos a partir de las cuales la herramienta construirá el sistema basándose en un modelo de colas.

Las arquitecturas consideradas representan diagramas de flujo de trabajos. A partir de éstos se creará automáticamente el esquema de servidores que distribuyan los trabajos de manera segura.

Se definirá una sintaxis que represente adecuadamente a los esquemas de procesos que serán considerados válidos por nuestro trabajo. Se debe denotar solamente aquellas arquitecturas de procesos válidos y la totalidad de las mismas.

Se considerarán válidas aquellas arquitecturas que representan un conjunto de procesos que cooperan para llevar a cabo una tarea común. La forma de cooperación que estudiaremos es la transformación de datos en sucesivas etapas llevada a cabo por procesos de distinto tipo, o como generalización, cualquier forma de trabajo cooperativa que requiera el pasaje de información entre procesos, en un cierto orden, para completar una tarea única en conjunto.

La arquitectura será representada mediante un grafo donde los nodos son procesos y por los ejes circulan datos. Los datos que ingresan al sistema y circulan por los ejes, son modificados en sucesivas etapas al pasar por los nodos del grafo.

En el primer nodo se modifican los datos de entrada, y dependiendo de características de los mismos, son derivados hacia uno u otro nodo del grafo. En cada nodo sufren transformaciones y circulan por los ejes hasta terminar de ser procesados en alguno de los nodos finales del grafo. La característica que determina hacia qué nodo se dirigirán los datos puede ser el grado de consistencia, distintos tipos de error de datos, tipo de transacción, u otra característica de relevancia para el flujo de trabajos representado por el grafo.

Esta cooperación de procesos puede representarse mediante una arquitectura de procesos con esquema de pipeline productor-consumidor. El concepto de pipelining se ve reflejado en las sucesivas transformaciones que se aplican a un mismo dato hasta obtener el dato completamente transformado que se genera en uno de los nodos finales. Cada proceso aplica una transformación a su dato de entrada y genera la salida que se convierte en la entrada del próximo proceso en el esquema reflejando la idea antes mencionada de productor-consumidor. El concepto de pipeline se aplica de una manera más general porque se permiten ciclos, ramificaciones y reducciones del pipe. Esta arquitectura de intercomunicación es representable mediante un grafo que cumple con las características de ser dirigido y con eventuales ciclos.

La visión más general de las arquitecturas representadas se refiere al conjunto de arquitecturas de procesos que interactuan para llevar a cabo un fin común, pasando

información -ya sean datos o control de estado- en cierto orden, entre los procesos. Permite a los desarrolladores comenzar con una visión abstracta del flujo de procesos y luego refinarla hacia un sistema implemenctado. La visión abstracta se define en términos de tareas a realizarse, informción pasada entre las tareas y la forma de rutear los trabajos entre las tareas. El modelo abstracto es un modelo del proceso que será soportado por la arquitectura.

Las arquitecturas descriptas pueden graficarse como un grafo donde cada P_i representa un proceso que realiza una operación sobre los datos que circulan por los ejes.

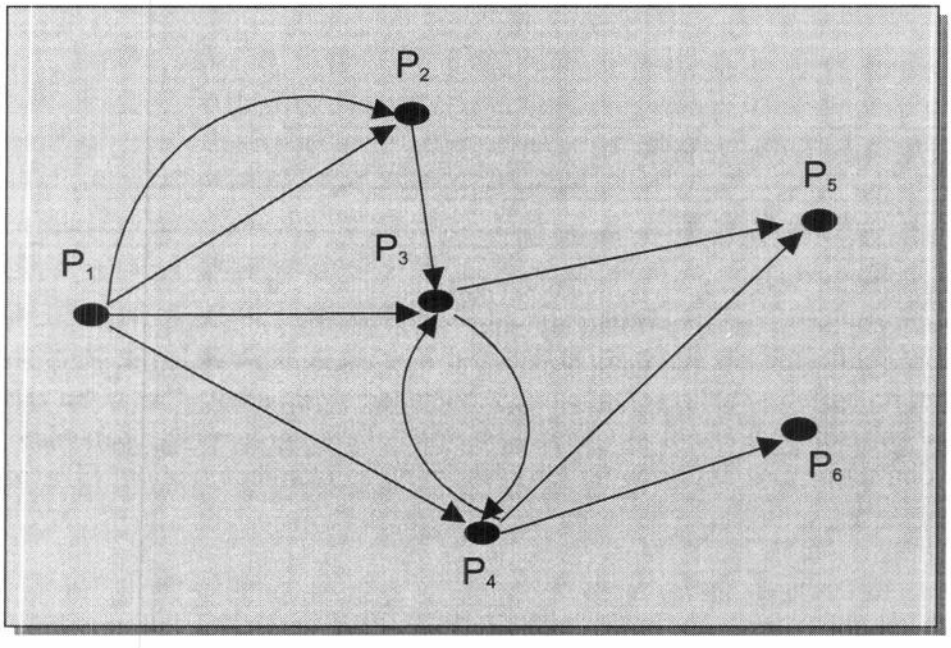


Figura 1 : Ejemplo de esquema de procesos

A continuación se enunciarán operaciones y reglas con el objetivo de organizar la descripción de las arquitecturas válidas. Los elementos considerados en las operaciones y reglas serán llamados *procesos*, *tokens* y *colas*. Los *procesos* son los procesos que conforman el sistema, el *token* representa un conjunto de datos que deberá ser procesado por un *proceso* constituyendo una unidad de trabajo para el mismo. Las *colas* son los repositorios temporarios de los *tokens*. La cola puede asociarse a un tipo de proceso o a un tipo de token (conjunto de tokens con determinadas características), siendo esto altamente dependiente de la semántica que se le asocie a una determinada arquitectura de procesos representable por nuestro esquema.

1.2.1 Operaciones válidas

1- Crear

La operación *crear*, introduce un nuevo token en el proceso que realiza la operación *crear*. Este token no existía anteriormente en el esquema de procesos.

2- Tomar

La operación *tomar* adquiere un token existente en la cola especificada y lo pasa al proceso que realiza la operación *tomar*. El token tomado ya existía en el esquema y debe pertener a la cola mencionada.

3- Poner

La operación *poner* opera sobre un token existente en el proceso que realiza la operación *poner* y lo pasa a la cola especificada. El token ya existía en el esquema y debe ser poseído por el proceso mencionado que previamente realizó la operación *tomar* sobre el token.

4- Borrar

La operación *borrar*, elimina un token ya existente en el esquema de procesos. El proceso que realiza la operación *borrar*, previamente realizó una operación *tomar* del token. Este token existía anteriormente en el esquema de procesos y luego de la operación *borrar* deja de ser parte del esquema.

1.2.2 Reglas del esquema

Regla 1

Si existe una cola en el esquema, existe al menos un proceso que realiza la operación *poner* en esa cola y al menos un proceso que realiza la operación *tomar* de la cola. Por lo tanto, no habrá una cola en la que nadie coloque tokens, y siempre esté vacía, ni habrá una cola de la que nadie tome tokens y por lo tanto se llene indefinidamente.

Regla 2

Si existe un proceso que realiza la operación *poner* colocando un token que tenía en su poder, entonces obtuvo ese token de una cola -mediante la operación *tomar*-, o creó el

token -mediante la operación *crear*-. Esto significa que ningún proceso puede hacer un *poner* de un token que no existía en el esquema y que no hubiera tomado previamente.

Regla 3

Si existe un proceso que realiza la operación *tomar,* obteniendo un token que no tenía en su poder, entonces colocará ese token en una cola -mediante la operación *poner*- o borrará el token -mediante la operación *borrar*-. Esto significa que ningún proceso hará acopio de tokens en forma indefinida.

Regla 4

Existe al menos un proceso que realiza la operación *crear* por esquema. Esto significa que existirán tokens en el mismo. No sería útil un esquema donde no existieran tokens para ser distribuidos.

Regla 5

Existe al menos un proceso que realiza la operación *borrar* por esquema. Esto significa que se eliminarán los tokens del mismo. No sería útil un esquema donde se introdujeran infinitos tokens para ser distribuidos y jamás se finalizara de procesarlos.

2. Definición de un lenguaje formal

En esta sección se introduce un lenguaje simple para representar las arquitecturas incluidas en el modelo descripto.

Se definirá un lenguaje que represente adecuadamente a los esquemas de procesos que serán considerados válidos por nuestro trabajo

Existe más de una manera de definir un lenguaje. Se eligió la siguiente sintaxis para definir un lenguaje que corresponda a las operaciones y reglas antes enunciadas y que describen las arquitecturas válidas para nuestro trabajo.

2.1 Gramática regular

Esq	→	Cadena / Cadena Esq
Cadena	→	P Resto
Resto	→	C P / C P Resto
P	→	P_1 / P_2 /.... / P_n
C	→	C_1 / C_2 /.... / C_m

Por motivos de implementación del analizador léxico y el parser se necesitará una gramática no recursiva a izquierda, por lo tanto presentamos la siguiente gramática que cumple esta característica

Esq	→	Cadena MasCadenas
MasCadenas	→	Cadena MasCadenas / λ
Cadena	→	P Resto
Resto	→	C P Sigue
Sigue	→	C P Sigue / λ
P	→	P_1 / P_2 /.... / P_n
C	→	C_1 / C_2 /.... / C_m

Estas dos gramáticas son equivalentes entre sí y también equivalen a la siguiente:

2.2 Expresión regular

$(P(CP)^+)^+$

2.3 Relación del lenguaje con las operaciones y reglas

En la siguiente sección veremos que la gramática propuesta cumple las operaciones y reglas enunciadas en el punto anterior.

La gramática genera las cadenas de la forma PCPCPCPC.....P. Un esquema reconocido será aquel formado por 1 o más cadenas generadas por esta gramática. Los procesos (P) y las colas (C) de igual nombre que se hallen en más de una cadena, colapsan en un único elemento del grafo. De esta manera un conjunto de cadenas genera un esquema con forma de red o grafo.

Operación *crear*:

Esta operación está representada por el primer P de cada cadena generada por la gramática.

La cadena $P_1C_1P_2C_2P_3C_3P_4$ significa que el proceso P_1 realiza la operación *crear* de los tokens para ese proceso.

Vemos que la operación mencionada está representada por la gramática presentada.

Operación *tomar*:

Esta operación está representada en la gramática propuesta por el par C_iP_j dentro de una cadena, significando que el proceso P_j toma tokens de la cola C_i.

La cadena $P_1C_1P_2C_2P_3C_3P_4$ significa que el proceso P_2 toma tokens de la cola C_1, el P_3 de la cola C_2 y el P_4 de la cola C_3.

Vemos que la operación mencionada está representada por la gramática presentada.

Operación *poner*:

Esta operación está representada en la gramática propuesta por el par P_iC_j dentro de una cadena, significando que el proceso P_i pone tokens en la cola C_j.

La cadena $P_1C_1P_2C_2P_3C_3P_4$ significa que el proceso P_1 pone tokens en la cola C_1, el P_2 en la cola C_2 y el P_3 en la cola C_3.

Vemos que la operación mencionada está representada por la gramática presentada.

Operación *borrar*:

Esta operación está representada por el último P de cada cadena generada por la gramática.

La cadena $P_1C_1P_2C_2P_3C_3P_4$ significa que el proceso P_4 realiza la operación *borrar* de los tokens para ese proceso.

Vemos que la operación mencionada está representada por la gramática presentada.

Regla 1:

Esta regla significa que si existe una cola en el esquema, existe al menos un proceso que realiza la operación *poner* en esa cola y al menos un proceso que realiza la operación *tomar* de la cola.

Las cadenas generadas son de la forma $P(CP)^+$ expresando que siempre hay una cola por cada cadena del esquema y que por cada C*i* que aparece en la cadena existe un P*j* inmediatamente anterior a la C*i* y otro P*k* inmediatamente posterior a la C*i*.

Como se vio anteriormente, P_jC_i significa que P_j realiza la operación poner en C_i y C_iP_k significa que P_k realiza la operación tomar de C_i.

Vemos que la regla mencionada está representada por la gramática presentada.

Regla 2:

Esta regla significa que si existe un proceso que realiza la operación *poner* colocando un token que tenía en su poder, entonces obtuvo ese token de una cola (mediante la operación *tomar*) o creó el token (mediante la operación *crear*).

Si el proceso realiza un *poner* significa que no es el último de la cadena ya que debe existir una cola (C) a la derecha en la cadena. Podría ser:

- El primer proceso de la cadena: sería una cadena de la forma de $P_1C_1P_2$.... donde P_1 es el proceso que consideramos para esta regla. Como hemos mostrado para las operaciones, P_1 realiza una operación *crear*.

- Un proceso interior a la cadena: la cadena sería de la forma $P_1C_jP_iC_i$ ... P_n donde P_i con $1<i<n$ es el proceso que consideramos para esta regla. Como hemos mostrado para las operaciones, P_i realiza una operación *tomar* de la cola C_j con $j=i-1$.

Vemos que la regla mencionada está representada por la gramática presentada.

Regla 3:

Esta regla significa que si existe un proceso que realiza la operación *tomar* obteniendo un token que no tenía en su poder, entonces colocará ese token en una cola (mediante la operación *poner*) o borrará el token (mediante la operación *borrar*).

Si el proceso realiza un *tomar* significa que no es el primero de la cadena ya que debe existir una cola (C) a la izquierda en la cadena. Podría ser :

- El último proceso de la cadena: la cadena sería de la forma de $P_1...C_nP_n$ donde P_n es el proceso que consideramos para esta regla. Como hemos mostrado para las operaciones, el proceso P_n realiza una operación *borrar*.

- Un proceso interior a la cadena: la cadena sería de la forma $P_1C_1P_iC_i$ ... P_n donde P_i con $1<i<n$ es el proceso que consideramos para esta regla. Como hemos mostrado para las operaciones, P_i realiza una operación *poner* en la cola C_i.

Vemos que la regla mencionada está representada por la gramática presentada.

Regla 4:

Esta regla significa que existe al menos un proceso que realiza la operación *crear* por esquema. Es trivial que cada cadena generada por la gramática propuesta comienza con un P_i . Como hemos mostrado para las operaciones, al ser P_i el primer elemento de la cadena realiza una operación *crear*.

Vemos que la regla mencionada está representada por la gramática presentada.

Regla 5:

Esta regla significa que existe al menos un proceso que realiza la operación *borrar* por esquema. Es trivial que cada cadena generada por la gramática propuesta termina con un P_i . Como hemos mostrado para las operaciones, al ser P_i el último elemento de la cadena realiza una operación *borrar*.

Vemos que la regla mencionada está representada por la gramática presentada.

Con el lenguaje propuesto existe más de una manera de representar una arquitectura, pero cada expresión del lenguaje denota a una única arquitectura. Una expresión del lenguaje que representa a la arquitectura del grafo de la Figura 1 es el siguiente conjunto de cadenas:

$P_1C_1\ P_2C_7P_3C_8P_4C_5P_5$
$P_1C_2\ P_2C_7P_3C_5P_5$
$P_1C_3P_3C_5P_5$
$P_1C_4P_4C_6P_6$
$P_1C_3P_3C_8P_4C_3P_3C_5P_5$

2.4 Especificación del lenguaje gráfico

En esta sección se presenta la definición de un lenguaje gráfico para expresar las arquitecturas soportadas. El lenguaje gráfico aquí presentado es equivalente a las gramáticas y expresión regular ya estudiados.

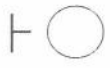
La mejor manera de describir esta arquitectura sería mediante una herramienta gráfica que tomara como entrada la especificación de una arquitectura en el lenguaje gráfico aquí definido y generara la especificación de la herramienta para manejo de los procesos y trabajos.

El lenguaje gráfico definido está constituído por la representación gráfica de las operaciones y reglas. Los tokens no se grafican en las operaciones y reglas, ya que cada una de las operaciones denota una acción unívoca sobre un token.

Un lenguaje gráfico es más llamativo y fácil de usar. La definición y la implementación no tienen complejidad conceptual. El motivo por el cual la implementación de la herramienta gráfica no se incluye en el presente trabajo de tesis radica únicamente en el tiempo de implementación que la misma conlleva.

2.4.1 Operaciones

1- Crear



2- Tomar

La operación *tomar* adquiere un token existente en la cola de la izquierda de la flecha y lo pasa al proceso en la derecha de la flecha.

3- Poner

La operación *poner* adquiere un token existente en el proceso de la izquierda de la flecha y lo pasa a la cola en la derecha de la flecha.

4- Borrar

La operación *borrar*, como su nombre lo indica elimina un token ya existente en el proceso que realiza la operación.

2.4.2 Reglas del esquema

1- Si ∃  entonces

2- Si ∃  entonces

o bien

3- Si ∃  entonces

o bien

4- ∃ al menos un (*crear*) ⊢

5- ∃ al menos un (*borrar*) ⊣

La representación de las arquitecturas soportadas por la herramienta serían grafos que cumplan las reglas y de manera que la estructura de los mismos determine las operaciones válidas para los tipos de proceso representados sobre las colas.

En la Figura 2 se representa la expresión del lenguaje gráfico que define la arquitectura de la Figura 1.

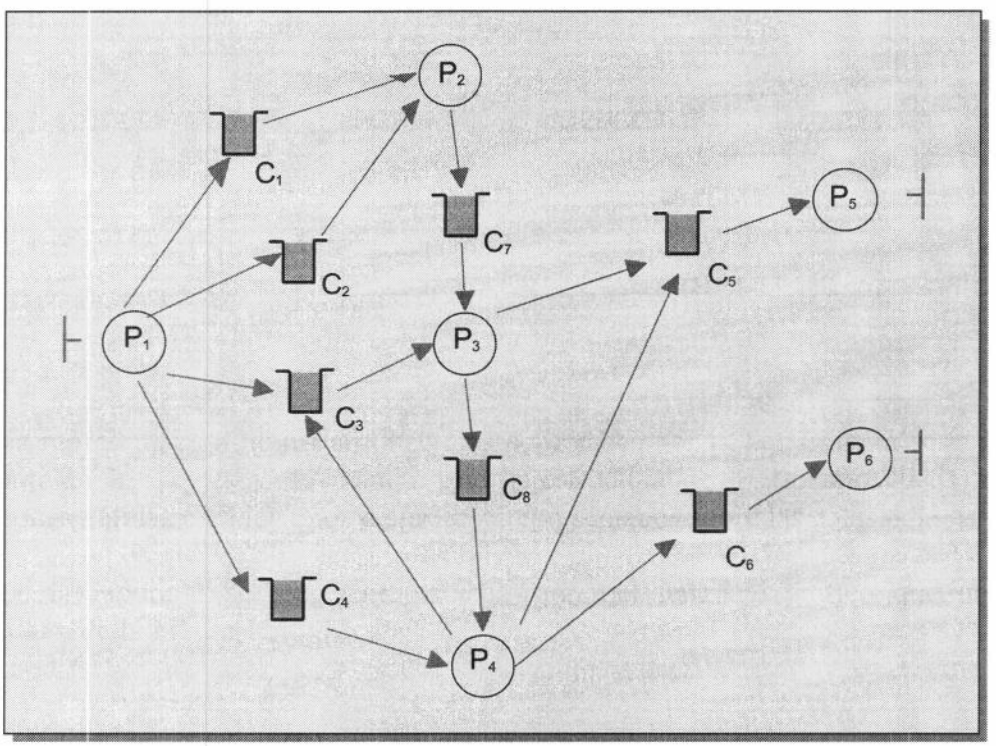


Figura 2: Lenguaje gráfico para el esquema de procesos ejemplo

3. Herramienta para definir arquitecturas.

En la presente sección se describe la implementación de una herramienta que soporte la definición de arquitecturas mediante el lenguaje anterior. Llamaremos configurador a esta herramienta.

El configurador define un configuración a partir del esquema de procesos y colas especificado en el lenguaje detallado. Esta configuración será la entrada para la herramienta tema del presente trabajo de tesis que manejará el flujo de trabajos para el esquema de procesos y colas especificados en el lenguaje.

3.1 Diseño del configurador.

El objetivo del configurador es leer una especificación escrita en el lenguaje y construir un archivo de configuración con toda la información necesaria para que la herramienta pueda administrar el flujo de trabajos en el conjunto de procesos especificados. A partir de la información escrita en el archivo de configuración, la herramienta creará los servidores que reconocerá y administrará el esquema de procesos y colas especificados en el lenguaje.

Se considera como una premisa de todo este trabajo que la herramienta final implementada tenga características que le permitan ser utilizada en un entorno real de trabajo. Por este motivo se diseñaron dos versiones del configurador.

En la primera versión se utilizaron las herramientas de Unix *lex* y *yacc* para construir un analizador léxico y un compilador simples para nuestro lenguaje.

La segunda versión está orientada al usuario final. De utilizarse la herramienta en un ambiente real de trabajo se realizará mantenimiento y administración sobre la arquitectura de procesos, pudiendo producirse modificaciones de diversa importancia a la misma. Si sólo se contara con el analizador léxico y compilador de la primera versión se debería construir el archivo de configuración a partir de una especificación en el lenguaje ante cada modificación. Para proveer un mecanismo más amigable al usuario final que no posea conocimiento detallado de lenguajes, se realizó una versión del configurador interactiva y con menúes en pantalla. Esta versión del configurador permite realizar altas, bajas y modificaciones de los objetos que integran la arquitectura de una manera más amigable y humana, mostrando mensajes explicativos en caso de errores y pidiendo confirmación ante modificaciones o borrado de objetos. Para poder ser ejecutada desde una terminal no gráfica se implementó usando cursores de Unix.

La salida del configurador es un archivo de configuración leído por la herramienta al comenzar su ejecución. El tener la posibilidad de ejecutar el configurador simultáneamente con la herramienta permite un muy rápido cambio en la configuración del esquema: sólo debe terminarse la ejecución de la herramienta y recomenzarse inmediatamente usando el nuevo archivo de configuración. Esto permite la realización de tareas de mantenimiento y administración del esquema sin interferir con la operación.

Si los cambios fueran profundos, también deberían considerarse las modificaciones necesarias a los repositorios de datos y a los datos en sí, cuestiones fuera del alcance de la herramienta y relacionados directamente con la semántica asociada a los procesos administrados por la misma.

3.1.1 Estructura de datos.

Las estructuras de datos utilizadas por el configurador en sus dos versiones son un subconjunto de las estructuras de datos usadas por los servidores que integran la herramienta. El configurador ejecuta en forma independiente a la misma y por lo tanto crea estructuras de datos idénticas a las usadas por la herramienta, pero en su propio espacio de memoria. Se diseñó de esta forma para permitir realizar tareas de mantenimiento y administración del esquema sin interferir con su operación.

3.1.2 Diseño e implementación del configurador.

El configurador en sus dos versiones ejecuta como un proceso independiente a los procesos que componen la herramienta. Es un proceso Unix normal que no es un servidor de RPC ni ejecuta como un demonio (en contraposición a los procesos de la herramienta que se estudiarán en el capítulo 5).

La primera versión del configurador formada por un analizador léxico y un compilador toma su entrada de un archivo y genera otro como salida.

La versión interactiva, permite la interacción con el operador de la herramienta a través de la terminal y presentando un esquema de menúes basados en cursores. También lee un archivo de entrada y genera otro como salida.

3.1.2.1 Primera versión: analizador léxico y compilador.

Para esta versión del configurador se utilizaron las herramientas lex y yacc que provee Unix para generar automáticamente un analizador léxico y un compilador para el lenguaje. Se escribió un proceso que utiliza el *lex* y *yacc* para generar el archivo de configuración usado como entrada por la herramienta.

En las distintas versiones de Unix las herramientas lex y yacc varían bastante, por lo tanto se preparó esta versión del configurador para poder ser ejecutada en DOS, OS/2 y Linux utilizando flex y yacc y en otros Unix (aunque sólo fue probado en AIX) usando las versiones generales de lex y yacc. Se necesita recompilar el configurador para cada uno de los entornos mencionados. La versión que presentamos en el presente trabajo de tesis ejecuta en Linux.

El configurador es un único proceso integrado por el analizador léxico, el compilador y la rutina que los invoca y genera el archivo de configuración.

3.1.2.1.1 Analizador léxico.

El analizador léxico es generado por la herramienta *flex* (o *lex*).

Se especifican cuáles son los tokens (aquí utilizado como término del lenguaje, no trabajo para la herramienta) aceptados por el lenguaje para representar un proceso o cola en el esquema. Los nombres de los procesos comienzan con el prefijo "PROC" o "proc" seguidos de una cadena alfanumérica que represente el nombre del proceso. Los nombres de las colas comienzan con el prefijo "COLA" o "cola" seguidos de una cadena alfanumérica que represente el nombre de la cola.

3.1.2.1.2 Compilador.

La herramienta *yacc* genera el compilador a partir de la gramática que representa el lenguaje. Se utiliza la gramática introducida en "Definición de un lenguaje formal" para especificar las acciones a realizar al recibir los tokens (aquí utilizado como término del lenguaje, no trabajo para la herramienta) reconocidos por el analizador léxico.

3.1.2.1.3 Rutina que invoca y genera el archivo de configuración.

La función main de esta versión del configurador se encarga de leer el archivo que contiene la expresión del lenguaje que representa el esquema a configurar y pasárselo al compilador. Luego inicializa las tablas donde se almacenarán los datos e invoca al compilador. Por último, escribe el archivo de configuración.

3.1.3 Ejemplo de archivo de configuración.

El siguiente es un ejemplo de archivo de configuración resultado de la ejecución del configurador sobre una expresión del lenguaje que define una arquitectura de procesos y colas válida. Este ejemplo representa la salida del configurador para la expresión de la Figura 1 y 2.

```
cantcolas= 8, cantproctype= 6
colarechazo=
BeginCola
C1,Harcodeado,805306372,805306373
C2,Harcodeado,805306374,805306375
C3,Harcodeado,805306376,805306377
C4,Harcodeado,805306378,805306379
C5,Harcodeado,805306380,805306381
C6,Harcodeado,805306382,805306383
C7,Harcodeado,805306384,805306385
C8,Harcodeado,805306386,805306387
EndCola
BeginProceso
P1
```

```
 s: C1, C2, C3, C4
fin
P2
 e: C1, C2, s: C7
fin
P3
 e: C3, C7, s: C5, C8
fin
P4
 e: C4, C8, s: C3, C5, C6
fin
P5
 e: C5
fin
P6
 s: C6
fin
EndProceso
```

3.1.3.1 Segunda versión : interactiva.

Esta versión del configurador está orientada al usuario final. Permite realizar altas, bajas y modificaciones de los objetos que integran el esquema de una manera más amigable, mostrando mensajes explicativos en caso de errores, pidiendo confirmación ante modificaciones o borrado de objetos, y otras características de aplicaciones orientados a usuarios finales. Para poder ser ejecutada desde una terminal no gráfica se implementó usando cursores de Unix.

Es un proceso Unix que interactúa con el operador a través de una terminal de caracteres. Se eligió una terminal no gráfica para permitir la ejecución del configurador desde cualquier tipo de terminal, inclusive por medio de un telnet a una terminal remota.

La funcionalidad de esta versión incluye la anterior y agrega la posibilidad de realizar modificaciones de diversa índole a un esquema ya existente mediante el uso de menúes de ABM de los objetos y las propiedades de los mismos.

El configurador se encarga de inicializar las tablas, leer el archivo de configuración existente e inicializar las variables utilizadas por los cursores para interactuar con la terminal. Luego presenta el menú principal. Al finalizar la interacción con el usuario, escribe el nuevo archivo de configuración.

Las funciones soportadas por el menú principal son:

1. Especificar una cola de rechazo.
2. Altas, bajas y modificaciones de Colas.
3. Altas, bajas y modificaciones de Procesos.
4. Salir.

Las altas, bajas y modificaciones de colas y procesos verifican que las modificaciones al esquema mantengan la validez de la arquitectura. Por ejemplo: una cola a eliminar no debe ser cola de entrada o salida para ningún tipo de proceso existente. Si se

deseara eliminar una cola de estas características se debería modificar el tipo de proceso para que no la use y luego eliminar la cola del esquema.

4. Diseño de la herramienta para administrar la arquitectura descripta por el lenguaje.

En la presente sección se presenta el diseño de la herramienta para construir automáticamente la estructura de alto nivel que permita la adminsitración de una arquitectura descripta por el lenguaje. En lo subsiguiente llamaremos "managers" al conjunto de servidores que conforman la herramienta.

4.1 Conceptos utilizados en el diseño de la herramienta

El sistema consta de colas de trabajos que son manejadas por los managers. Estos actúan como distribuidores de trabajos, garantizando no entregar el mismo trabajo a más de un cliente y evitando la pérdida de trabajos ante posibles caídas.

Algunos conceptos claves:

Tipo de proceso: Cada cliente que desee conectarse con el manager debe hacerlo como algún "tipo de proceso" predefinido. El tipo de proceso especifica a los managers los permisos y accesos de los diferentes procesos a los recursos del sistema, y por lo tanto a las diferentes colas y trabajos. Estos tipos de proceso serán especificados por el administrador del sistema. El administrador le informará al programador que "tipo de proceso" debe usar, dependiendo de las colas que usará de entrada y de salida. Si un proceso desea conectarse con una cola no permitida para su correspondiente tipo de proceso, se retornará error.

Cola: Las colas son los repositorios temporales de los trabajos. A cada tipo de cola se asocia un trabajo en un estadio distinto de procesamiento o un tipo de dato o información. Cada proceso que desee transformar un trabajo lo toma de alguna de sus colas de entrada, lo modifica y lo coloca en alguna de sus colas de salida. Dependiendo de la arquitectura de la aplicación a cada cola se asocian el mismo o diferentes tipos de datos.

4.2 Características de la arquitectura de procesos y propiedades de la herramienta

En la presente sección se enumeran algunas características de la arquitectura de procesos que influencian el diseño de la herramienta. Muchas de las características enunciadas se aplican en general a sistemas distribuidos y han sido tratadas en mayor detalle en la sección 8, Anexo A: Características de los sistemas distribuidos en página 149. También se enumeran propiedades que deberá cumplir la herramienta para poder

administrar el conjunto de procesos que se describe, manteniendo características también introducidas a continuación.

- Los procesos de la arquitectura ejecutan en un entorno distribuido. Reciben datos, los transforman y pasan el dato transformado, en general existe cooperación entre procesos mediante el traspaso o intercambio de información.

- Se permiten múltiples procesos idénticos ejecutándose simultáneamente. Diremos que todas esas instancias corresponden al mismo *tipo de proceso*.

- Entre los distintos tipos de proceso existe una relación con respecto a los datos que comparten: es requisito que la entrada para un proceso de un tipo haya sido la salida de otro tipo de procesos. Esto denota cooperación ordenada para cumplir un objetivo común.

- La cantidad de instancias, procesos de un determinado tipo, que estan en ejecución en un determinado momento es dinámica, limitando la facilidad de definir sus características estáticamente.

- No se pueden perder datos ante caídas de la red, de los procesos u otro tipo de falla.

- Se debe garantizar que dos instancias de un mismo tipo de proceso, no estén trabajando con el mismo dato simultaneamente. Esto podría ocasionar inconsistencias y/o redundancias en los resultados finales de procesamiento. Garantizar la unicidad del dato con respecto a las distintas instancias de un mismo tipo de proceso.

- El flujo del dato puede no ser lineal, observándose ciclos en el transcurso de las transformaciones realizadas al mismo.

- No habrá límites de tiempo en la ejecución de los distintos procesos.

- Al carecer de sincronización temporal entre los distintos procesos, es necesario implementar un mecanismo de buffering entre ellos para equilibrar las distintas velocidades de ejecución entre los procesos productores y consumidores. En caso que un cliente requiera un nuevo trabajo para continuar con su proceso y no haya uno disponible, se debe poder de bloquear al cliente y esperar que llegue un trabajo antes de devolverle el control. Se deben proveer dos modalidades: bloqueante y no bloqueante.

- Debe mantenerse consistente el estado de los datos al ingresar y permanecer en dichos buffers, hasta que sean retirados. Debe utilizarse un almacenamiento perdurable para guardar los trabajos.

- Al existir múltiples máquinas ejecutando procesos para un fin común, se debe evitar que la caída de alguna de ellas o de algún proceso y su posterior recuperación perjudique el comportamiento del conjunto y la consistencia de los datos. Se debe ser capaz de conectar los distintos tipos de procesos y registrarlos de manera tal de poder mantener un registro de su funcionamiento con el objetivo de actuar en caso de caída de alguno de ellos.

- Deben garantizar la posibilidad de realizar rollback ante la caída de un cliente. El rollback consiste en no perder el/los trabajo/s sobre los que estaba actuando el proceso cliente en el momento de la caída y la redistribución de estos trabajos a otros clientes del mismo tipo. No se deben perder trabajos.

- Los procesos pueden ejecutar en distintas plataformas.

- Los procesos intercambian mensajes vía un protocolo de comunicación común.

- Enriquecer las herramientas provistas para la comunicación con mecanismos que aseguren alta confiabilidad.

- El esquema del grafo de procesos puede ser trasparente para cada uno de ellos. Cada proceso sólo conoce con quien interactúa directamente. Puede no tener conocimiento del flujo completo de los datos, ni debe preocuparse por la consistencia de los mismos.

- Proveer herramientas para definir la arquitectura y hacer que se interprete esta información y se implemente el control global que necesita el esquema para poder llegar correctamente al objetivo descripto.

- Es necesaria una herramienta que permita alterar el flujo normal de trabajos en cualquier momento, con el objetivo de administrar el mismo manualmente.

- Este esquema debe responder eficientemente a grandes volúmenes de trabajos.

- En función de los tiempos de desarrollo que manejemos se podría realizar una herramienta para obtener información estadística sobre el funcionamiento de los diferentes procesos para poder ser analizada con el objetivo de mejorar la carga del sistema y flujo de la información en el esquema de procesos dado.

- Dada la característica de la falta de conocimiento global y nuestra necesidad de tener información del estado del conjunto de procesos pertenecientes al esquema de grafo, se deberá proveer un mecanismo que brinde esta información, permitiendo la actuación conjunta y cooperativa de los procesos. Por lo tanto, la herramienta debe ser capaz de conectar los distintos procesos y registrarlos de manera tal de poder mantener un registro de su funcionamiento y actuar en caso de caída de alguno de ellos.

- En caso que una de las colas esté vacía, y un cliente requiera un nuevo trabajo para continuar con su proceso, la herramienta debe ser capaz de bloquear al cliente (y a sí mismo) y esperar que llegue un trabajo antes de devolverle el control al cliente, evitando así un busy waiting. Se debe proveer también la funcionalidad no bloqueante por si fuera requerida por la semántica asociada al esquema de procesos administrado.

4.3 Consideraciones para el diseño de la herramienta

Las siguientes características serán tenidas en cuenta durante el diseño de la herramienta. El subconjunto de características descriptas en la presente sección se basa en definiciones recopiladas en [ACM 031] y en conceptos extraídos de [Peterson 010], [Singhal 003], [Mullender 001]. Estas características están relacionadas con los aspectos más generales de sistemas distribuidos tratados en la sección 8, Anexo A: Características de los sistemas distribuidos en página 149.

4.3.1 Recuperabilidad

Recuperabilidad en sistemas de computación se refiere a la posibilidad de restaurar el sistema a su estado operacional normal. La recuperabilidad puede ser algo tan sencillo como reiniciar una máquina caída o un proceso que falló. En la mayoría de los casos, para volver a un estado válido al reiniciar la computadora o el proceso, se deben haber tomado medidas preventivas en la forma de ejecutar de los procesos a reiniciar. Existen varias técnicas para poder volver a un estado válido de ejecución, ya sea el mismo estado del momento de la caída o un estado válido anterior. Generalmente el diseño de sistemas o procesos que logren recuperabilidad es bastante complicado.

Alternativas para la recuperabilidad:

- Si un proceso ha producido modificaciones es importante que todas las modificaciones realizadas por el proceso que falló sean deshechas hasta llegar a un estado válido anterior.
- Si un proceso ha ejecutado por un largo tiempo antes de fallar sería preferible reiniciar el proceso desde el punto de su caída y continuar con su ejecución. De esta manera se evita tener que ejecutar el proceso desde su comienzo, lo que puede ser una operación costosa y pérdida de tiempo.

Hay que tener en cuenta en el caso de tareas que cooperan es que si alguna de ellas falla entonces los efectos debidos a la interacción del proceso caído deben ser deshechos, de otra manera cada uno de los procesos que cooperan debería ser reiniciado desde un estado apropiado.

Para nuestro diseño podemos estudiar el aspecto de recuperabilidad en 2 niveles conceptuales. En el primer nivel podemos considerar a los clientes que usan la herramienta, los procesos de la arquitectura, como un conjunto de procesos que cooperan para lograr un objetivo común. Se pretende conseguir ese fin y no permitir que la caída o falla de algunos de los procesos evite lograr el objetivo. Para recuperar la falla de un proceso, los managers proveen un mecanismo implementado por módulos especiales (monitor/watchdog) que monitorean el accionar de los clientes de la herramienta. Si un cliente falla los trabajos que estaban siendo procesados por el mismo se retornan a las colas a las que pertenecían antes de ser tomados por el cliente. Permitiéndose así la recuperación de la arquitectura de procesos ante la falla de uno o más de ellos.

El segundo nivel está representado por la herramienta en sí. No se ha implementado ningún mecanismo de recuperación de los managers en sí. Si alguno de los servidores falla, todo el conjunto de servidores termina su ejecución. No es posible continuar la ejecución de los managers de una manera segura y consistente si alguno de sus componentes falla o muere. Existe un mecanismo implementado mediante señales que permite la terminación de los managers en caso de muerte de cualquier servidor de los mismos. Los clientes de la herramienta, al tratar de realizar alguna operación contra los managers, recibirán un mensaje de error que informe de la caída. Al recomenzarse los managers se retorna al estado consistente inmediatamente previo a la caída. Como se estudiará más adelante, todas las operaciones son atómicas y por lo tanto todos los estados del sistema son consistentes.

4.3.2 Tolerancia a Fallas

Al hablar de recuperabilidad se mencionaron técnicas para recuperarse de caídas y fallas, sin embargo las interrupciones causadas durante las caídas pueden ser severas en algunas casos, como por ejemplo sistemas online o procesos de control.

Para evitar las interrupciones en el sistema debidas a fallas y mejorar la disponibilidad se puede optar por diseñar los sistemas para ser tolerantes a fallas.

Un sistema puede diseñarse para ser tolerante a fallas en dos sentidos:

1- **Enmascarar fallas:** continua realizando su función especifica en caso de una falla.

2- **Exhibir un comportamiento bien definido ante fallas:** Puede o no seguir realizando su función ante una falla, pero facilita acciones apropiadas para la recuperación. Un ejemplo de este caso es que las modificaciones realizadas por una transacción a una base de datos se hacen visibles a otras transacciones sólo en el caso que se realice el commit. En el caso que falle la transacción sin realizar el commit, sus modificaciones nunca fueran percibidas por las demás transacciones y por lo tanto no se afectó su ejecución.

Un factor clave en el logro de tolerancia a fallas es la redundancia. Se utiliza un número múltiple de procesos, hardware o copias de datos cada una de ellas con diferentes comportamientos ante fallas. Se utilizan diferentes protocolos en el diseño de sistemas tolerantes a fallas.

Como un sistema tolerante a fallas debe comportarse de una manera especifica en caso de una falla, es importante estudiar las implicaciones de ciertos tipos de fallas.

Muerte de procesos: Cuando un proceso muere es importante que los recursos a él asignados sean recuperados, de otra manera podrían perderse permanentemente. Cuando se ejecuta en un modelo cliente-servidor y se produce una falla en el servidor debe informarse a la máquina cliente para que el proceso esperando una respuesta pueda ser desbloqueado y tomar las medidas necesarias. De manera análoga, si el proceso cliente muere después de haber realizado el pedido de servicio es necesario que se le informe al servidor de la muerte del cliente. Esto permitirá al servidor reclamar los recursos que haya destinado al cliente caído.

Falla de la máquina: Todos los procesos ejecutando en la máquina morirán. En lo referente al comportamiento del cliente o el servidor no hay mayor diferencia entre la caída de una máquina o la muerte de un proceso. La única diferencia radica en como es detectada la falla. En el caso de muerte de procesos otros procesos permanecen activos (incluido el kernel del SO), por lo tanto es posible enviar un mensaje informando que un determinado proceso ha muerto a cualquier otro que pregunte. En caso de caída de la máquina se puede determinar que la ausencia de mensaje de respuesta significa la muerte del proceso que debe monitorear o responder y la consecuente caída de la máquina.

Falla de la red de comunicaciones: Una falla en un enlace de comunicación puede partir una red en subredes, haciendo imposible que una máquina en una subred se comunique con otra en una subred diferente. Un proceso no puede darse cuenta de la diferencia entre al caída de una máquina o de un enlace de comunicación, a no ser que la capa de comunicación de red pueda reconocer la falla de una máquina. (Por ejemplo: Token ring devuelve un código de error adecuado pero Ethernet no). Si la capa de comunicación de red no puede retornar un código de error adecuado, un diseño que cumpla con las características de tolerancia a fallas deberá asumir que la máquina está activa y los procesos operando.

La tarea de diseñar y entender arquitecturas de sistemas distribuidos que son tolerantes a fallas no es nada sencillo, ya que se debe no sólo estar en control de las actividades normales que desarrolla el sistema, sino que también de situaciones complejas que ocurren cuando fallan algunos componentes.

Hay tres conceptos que vale la pena aclarar : **servicios**, **servidores** y la **relación de dependencia** que existe entre los servidores. Un *servicio* computacional especifica un conjunto de operaciones cuya ejecución es disparada por entradas generados por usuarios

de este servicio o el paso del tiempo. Esta ejecución puede resultar en salidas para el usuario que realizó el requerimiento y/o en un cambio de estado del servidor.

Las operaciones definidas en una especificación de un servicio sólo pueden ser llevadas a cabo por un *servidor* para este servicio. Un servidor implementa un determinado servicio sin exponer a los usuarios los detalles de implementación ni los estados internos por los que pasa el servidor para poder satisfacer dicho servicio.

Hay servidores que basan la respuesta a un servicio utilizando a su vez otros servicios implementados en otros servidores. Se dice que un servidor ***u depende*** de un servidor ***r*** si la correctitud del comportamiento de ***u*** depende de la correctitud del comportamiento de ***r***. En este caso, el servidor ***u*** pasa a ser llamado "*usuario*" de ***r***, mientras que ***r*** pasa a ser llamado un "*recurso*" de ***u***.

Tanto las nociones usuario como recurso/servidor son relativas a la "relación de dependencia" existente: lo que es recurso o servidor en un cierto nivel de abstracción puede ser usuario en otro nivel de abstracción.

Nuestro esquema no pretende ser tolerante a fallas, ya que es independiente del hardware sobre el que se implemente, y por lo tanto, no tiene requerimientos de redundancia o disponibilidad. Consideramos, sin embargo, útil el estudio de sistemas tolerantes a fallas porque nuestro esquema considera las posibilidades de falla por muerte de proceso, falla de máquina y falla de red de comunicación y actua para mantener el sistema en estado consistente y recuperarse de fallas. Se han implementado a tal efecto mecanismos de monitoreo de procesos en máquinas distribuidas. Si un cliente falla los trabajos que estaban siendo procesados por el mismo se retornan a las colas a las que pertenecían antes de ser tomados por el cliente.

Clasificación de las Fallas

Un servidor diseñado para proveer un cierto servicio se dice que es ***correcto*** si, en respuesta a entradas, se comporta de manera consistente con lo especificado en el servicio. Asumimos que se encuentra especificado tanto la respuesta a una determinada entrada para cualquier estado inicial del servidor, así como el intervalo de tiempo real dentro del cual se debe generar dicha respuesta.

Ocurre una ***falla*** en el servidor si éste no se comporta de forma especificada. Una falla por ***omisión*** ocurre cuando un servidor "*se olvida*" de responder una entrada. Una falla de performance ocurre cuando la respuesta es funcionalmente correcta pero fuera del intervalo especificado para la misma. Una falla de ***respuesta*** ocurre cuando el servidor responde incorrectamente, ya sea porque el valor retornado es incorrecto o el estado de transición que toma lugar es incorrecto.

Si luego de una falla por omisión en generar una salida, el servidor continúa omitiendo salidas a subsecuentes requerimientos hasta que no se inicialice de nuevo, el servidor se dice que sufrió un ***crash failure*** (*Falla por caída*).

Dependiendo del estado del servidor una vez reinicializado, se puede distinguir entre diferentes tipos de comportamientos ante un *crash failure*.

Se dice que ocurre una **caída con amnesia** cuando el servidor levanta en un estado inicial predefinido que no depende de las entradas que ocurrieron antes de la caída. Es una **caída con amnesia parcial** cuando una parte del estado es la misma que se encontraba antes de la caída y otra parte del estado es vuelta a un estado inicial predefinido. También se habla de una **pausa-caída** que ocurre cuando el servidor reinicia su ejecución exactamente en el mismo estado en que se encontraba en el momento que se produjo la caída. Por último, tenemos un **halting-crash** que ocurre cuando el servidor no puede volver a reinicializarse.

Un ejemplo de caídas puede ser la caída de un sistema operativo seguida por un reboot que lo vuelve a un estado inicial del sistema ya predefinido, o la caída de un servidor de base de datos que se levanta con un estado igual al commit de todas las transacciones que se estaban llevando a cabo antes de la caída.

La caída de los managers representa una caída con amnesia parcial ya que se restauran los trabajos al estado en que deben estar y que no se pierde ninguna operación sobre los mismos. Se toman en cuenta las últimas entradas que se realizaron antes de la caída solamente referidas a los trabajos, pero la información sobre los clientes y las tablas en memoria se vuelven a un estado inicial predefinido, en el caso de la herramienta es vacío.

Tomamos en cuenta todos las entradas que ocurrieron antes de la caída siempre que resulten en un estado consistente. Un estado consistente implica aceptar eventuales repeticiones de trabajos. Supongamos el caso en que un cliente esté operando con un trabajo, ha terminado su ejecución pero no se puede comunicar con los managers debido a que los mismos fallaron. Al reinicializarse los managers, el trabajo sobre el que el cliente estaba trabajando va a pasar a un estado de disponible para el mismo tipo de procesos que lo tenía antes ya que el cliente no avisó que había finalizado con el trabajo. En este caso dos procesos van a hacer exactamente el mismo trabajo, aunque sólo uno de ellos tenga éxito. El servidor considerará sólo aquel que logre comunicar la finalización de la operación.

No hay requerimiento respecto a la conducta de los clientes. La caída de los clientes podría ser un caso de caída con amnesia ya que el cliente más simple de la herramienta es un proceso consistente de un ciclo continuo de procesamiento de trabajos: toma un trabajo, lo proceso y lo retorna al esquema. Un cliente de estas características no tienen nada para recordar luego de una caída. Simplemente vuelve a su ciclo. Como la semántica de los clientes dependerá de cada caso particular y no se incluye el estudio de los mismos en el presente trabajo, no pueden claisificarse las fallas de los mismos.

Semántica de las Fallas

Cuando se piensa en un esquema de recuperabilidad, es importante saber cuáles son los posibles comportamientos que puede tomar un servidor ante la ocurrencia de una falla. Un ejemplo que ilustre esta idea sería: supongamos que tenemos un cliente **cl** que envía un requerimiento **rq** a un servidor **sr** mediante un link **l**, el cliente sabe que la comunicación con el servidor via **l** sufre sólo de *falla por omisión* (no de performance lo que significa que si la respuesta no llega en un lapso predeterminado significa que no va a

llegar), entonces si **cl** no recibe una respuesta de **sr** via **l** en un intervalo de tiempo especificado, **cl** puede asumir que la respuesta a **rq** no va a llegar más, y puede tomar la decisión de enviar un nuevo requerimiento **rq'** a **sr** via **l**, pero no tendrá que mantener ninguna información adicional relacionada al requerimiento **rq**, que permita diferenciar entre una respuesta a **rq** o al requerimiento actual **rq'**, ya que ninguna respuesta a **rq** va tener que ser manejada dado que **sr** no sufre de falla por performance. De otra manera tendríamos que mantener información adicional para poder diferenciar cuando descartar o no una respuesta como duplicado.

Si la especificación de un servidor ***s*** describe que el comportamiento de ***s*** ante una falla que van a observar los usuarios del mismo debe ser de clase F, entonces se dice que ***"s tiene semántica de falla F"***.

Ejemplo de esto sería un servicio de comunicación al que se le permita la pérdida de mensajes pero la probabilidad de que tengan retrasos o vengan corruptos es casi nula, entonces podemos decir que este servidor tiene una semántica de falla por omisión. Por el contrario, si tenemos un servicio al cual se le permite retrasos y pérdidas de los mensajes, pero nunca que éstos estén corruptos, entonces decimos que tiene una semántica de falla por omisión/performance.

En general, cuanto más fuerte es la especificación de la semántica de la falla, más caro y complejo es la construcción de un servidor que la implemente. Ejemplo de esto es un procesador que resuelve una semántica de falla ante caídas mediante la duplicación y el matching, es mucho más caro que un simple procesador que no tiene ningún tipo de redundancia que evite que los usuarios vean un comportamiento arbitrario distinto al esperado, ocasionado por alguna falla.

La herramienta tendrá un módulo llamado watchdog/monitor que será el encargado de monitorear el funcionamiento de los clientes y reaccionar en caso de falla de los mismos. Tendría una semántica de falla por performance debido a que puede tardar, pero al utilizar RPC sobre TCP/IP como medio de transporte, tarde o temprano llega la respuesta y sin errores. Para los servidores que integran los managers contamos con una semántica de falla por caídas.

Enmascaramiento de Fallas Jerárquicas (Hierarchical Failure Masking)

Un comportamiento ante una falla, puede ser clasificado sólo con respecto a una especificación particular de un servidor, y a un cierto nivel de abstracción. Es decir, que si un servidor depende de otro de más bajo nivel para brindar correctamente un servicio, entonces una falla de un cierto tipo en un nivel bajo de abstracción puede resultar en una falla de otro tipo a otro nivel más alto de abstracción.

En este tipo de sistemas, el mecanismo de *exception handling* provee una manera conveniente de propagar información sobre la detección de fallas a través de los distintos niveles de abstracción y de enmascarar fallas de bajo nivel a servidores de alto nivel.

La herramienta posee jerarquías de servidores donde servidores de bajo nivel proveen servicios a servidores de más alto nivel. Este es el caso de los servidores que utilizan al logserver (módulo de los managers) como proveedor del servicio de logging interno o propio de los managers. No utilizamos un esquema de enmascaramiento de fallas, debido a que el tipo de fallas que pudieran ocurrir en los servidores de bajo nivel impiden que el resto de los servidores presten sus servicios. El buen funcionamiento del logserver es indispensable para que los managers conserven su característica de no perder trabajos. También se utilizan servicios del sistema operativo, como ser RPC, DNS, syslog, etc.

Si alguno de los servidores falla, todo el conjunto de servidores termina su ejecución. No es posible continuar la ejecución de los managers de una manera segura y consistente si alguno de sus componentes falla o muere.

Software Architectural Issues

El objetivo a alcanzar sería permitir que un servidor sea dado de baja de un sistema (debido a fallas) sin interrumpir la actividad de los usuarios. Cuando esto es imposible de lograr, el siguiente objetivo debería ser el de asegurar que estos servidores tengan un semántica de fallas "agradable", como ser crash, omisión o performance. Esto va a permitir que sus usuarios (posiblemente personas) puedan recuperarse ante las fallas, mediante un enmascaramiento como ser "hagan un login de nuevo" o "esperen por un rato y vuelvan a intentar".

Dependiendo si el estado de un servicio es persistente o no, se puede requerir que los servidores que realizan este servicio, provean una semántica de transacciones atómicas de fallas, o simplemente amnesia-total o amnesia-parcial en una semántica de falla por caídas. La idea de atomicidad es que se tiene que pasar de un estado consistente a otro estado consistente, si éste no se puede lograr, entonces no se realiza ningún cambio.

Por otro lado, para servidores de bajo nivel, como ser manejadores de I/O que no tienen un estado persistente, generalmente es satisfactorio que ofrezcan una semántica de fallas ante caídas con amnesia total, es decir, luego de una falla, estos servidores deben reinicializarse a sí mismos y aceptar nuevos requerimientos como si no hubiera pasado nada.

Hay un par de conceptos a tener en cuenta con respecto a los programas clientes que utilizan estos servicios. Se dice que un programa es ***totalmente correcto*** si se comporta de acuerdo a lo especificado ante cualquier entrada bajo la condición que los servicios que éste utilice no fallen. Asimismo, un programa ***parcialmente correcto*** puede sufrir un crash o performance failure con ciertos entradas aunque los servicios de los cuales dependa no hayan fallado.

Una consecuencia que este tipo de programas parcialmente correctos puede traer es que cuando el servidor genera un salida y una transición de estado en respuesta a un requerimiento de un usuario, este salida puede nunca alcanzar al cliente, dejando al

servidor en un estado consistente y correcto si se hubiese entregado la respuesta, pero no en otro caso.

Alcanzar una correctitud parcial es más fácil de lograr, debido a que una correctitud total implicaría tener inmunidad a todo tipo de entradas. Además, hoy en día encontrar un software que tenga caídas o sea lento, pero que nunca entregue resultados corruptos, es bastante común. Por último, tener un servidor parcialmente correcto, que por definición significa que tiene semántica de fallas crash/performance mientras que sus servidores (nuestro servidor pasaría a ser cliente) de más bajo nivel nunca fallen, mantiene esta misma semántica aunque los servidores de más bajo nivel sí fallen.

La herramienta provee una semántica de transaciones atómicas porque mantiene un estado persistente. En nuestro esquema, cuando un cliente hace un requerimiento puede ocurrir que el cliente muera y la respuesta del servidor no llegue a nadie. El estado ya cambió (Los managers cambiaron el estado del trabajo a tomado o bien lo cambiaron de cola). El servidor debe enterarse y volver a un estado consistente (retornar el trabajo a su cola original). La herramienta utiliza al watchdog/monitor como mecanismo para descubrir fallas en los clientes. Los clientes de la herramienta son parcialmente correctos.

5. Diseño de los servidores que conforman los managers

5.1 Introducción al esquema de procesos que conforman los managers

La arquitectura especificada por el lenguaje es interpretada por los managers antes de formar su estructura. La estructura de los mismos es dinámica y depende de la arquitectura del modelo para el cual los mismos serán utilizados. Las colas, tipos de proceso y demás elementos que intervienen en la arquitectura determinan los recursos a inicializar y utilizar por los managers.

Los managers son dinámicos en lo referente a la alocación de recursos, pudiendo soportar arquitecturas de tamaños variados sin influir negativamente en la performance. El tamaño más adecuado para cada una de las estructuras de datos, medios de comunicación e instancia de servidores se determina en el momento de la inicialización de los managers.

Se determinó una relación entre las colas y los recursos del sistema. La cantidad de colas definidas es el principal parámetro para determinar la cantidad de recursos que se reservan para la ejecución.

Para cada configuración existirá un número diferente de instancias de servidores y distinta cantidad de recursos reservados, guardando una relación directa con la configuración que sirve de entrada a los managers.

La arquitectura de un esquema puede modificarse en momento de ejecución, pero los cambios tomarán efecto al reinicializarse los managers. Al leer la nueva configuración, se generarán las instancias de colas y servidores necesarios para los nuevos requerimientos. Los trabajos pendientes se seguirán conservando siempre y cuando existan las colas en los cuales se hallaban los mismos. El usuario debe tener especial cuidado de no cambiar una arquitectura de manera inconsistente con los datos almacenados en la misma.

5.1.1 Estructura de los procesos que conforman la herramienta

El servidor de trabajos o manager es un conjunto de módulos con la responsabilidad de administrar y garantizar consistencia en un sistema de manipulación de trabajos. Se utilizan colas como medio de comunicación entre procesos. Las colas constituyen el repositorio temporal de los trabajos a administrar por los managers.

Los managers administran procesos que trabajan en un ambiente distribuido. En contraposición a esto la filosofía de diseño de los mismos es la utilizada en sistemas de administración centralizada. Los managers son los encargados de repartir los trabajos a los procesos que se encuentran distribuidos en diferentes máquinas comunicadas a través de una red. Los managers centralizan la información y se encargan de solucionar los

problemas surgidos al tener que manejar los recursos compartidos por los procesos clientes.

Existe un servidor, el *connectserver*, encargado de crear todos los demás servidores que conforman los managers. Este módulo también recibe los pedidos de conexión de los clientes de los managers. Es el servicio de los managers conocido por nombre.

El diseño consta de varios servidores encargados de tomar trabajos de la cola y entregarlos, y de aceptar trabajos para encolar. Estos servidores mantienen una estructura de datos en común. Se utilizan mecanismos de sincronización para el acceso a las estructuras compartidas para evitar inconsistencias. Se asocian dos servidores, denominados *getserver* y *putserver*, a la administración de cada cola definida en la arquitectura. Se tendrán tantas instancias de *getserver* y *putserver* como colas se encuentren en el sistema. La comunicación entre las diferentes instancias de los procesos se realizará a través de mecanismos de memoria compartida y de semáforos.

Con el objetivo de recuperarse ante fallas, y llevar un registro global del estado del sistema en todo momento, se diseñaron dos servidores o módulos llamados *logserver* y *historyserver*. El primero recibe los registros de cada una de las transacciones que modifican el estado de los managers y las graba físicamente para poder ser recuperados en caso de caídas. El segundo módulo consolida y procesa toda la información que va registrando el módulo *logserver*. El *logserver* registra todas y cada una de las operaciones realizadas sobre los trabajos manejados por los managers, mientras que el *History* transforma la historia de los movimientos en el estado actual de trabajos y se eliminan del estado del sistema aquellos trabajos que han sido borrados. Se almacena solamente la última ubicación de un trabajo (en qué cola se encuentra).

Existe un módulo que se encarga del seguimiento y monitoreo de cada uno de los clientes para detectar fallas. Detectar una falla permite liberar el o los trabajos retenidos por el cliente para que puedan ser procesador por otras instancias. Esta tarea la cumplimentan un conjunto de módulos denominados *Monitor/Watchdog*.

Los módulos de los managers son los mostrados en la siguiente figura:

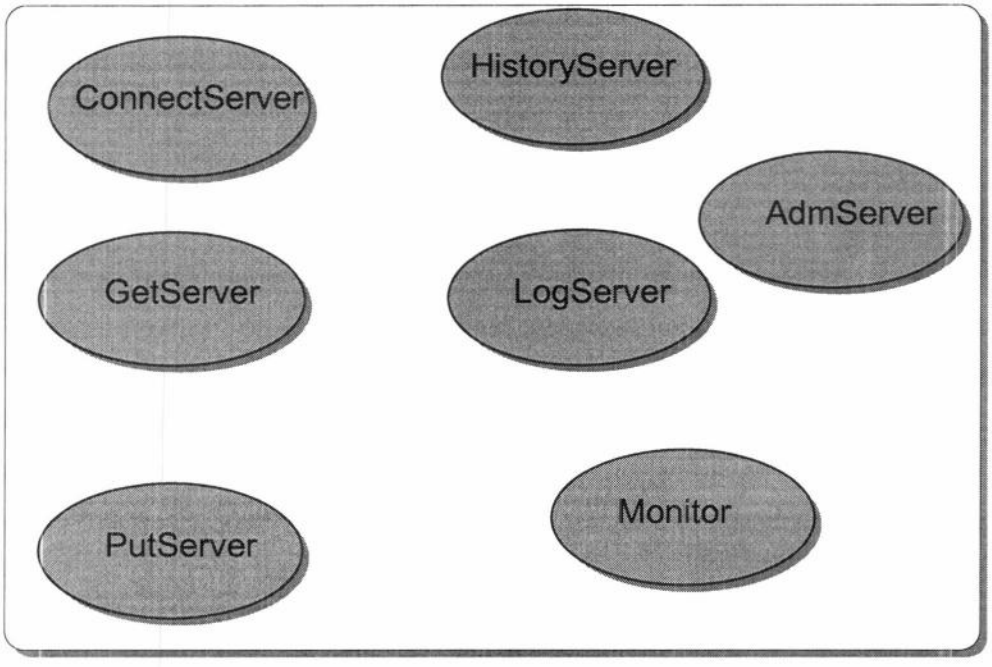


Figura 3 : Servidores que integran la herramienta

La siguiente figura muestra la relación padre-hijo de los procesos que constituyen los managers. Como se explicó anteriormente, el módulo Connectserver es el proceso padre de todos los demás procesos de la herramienta.

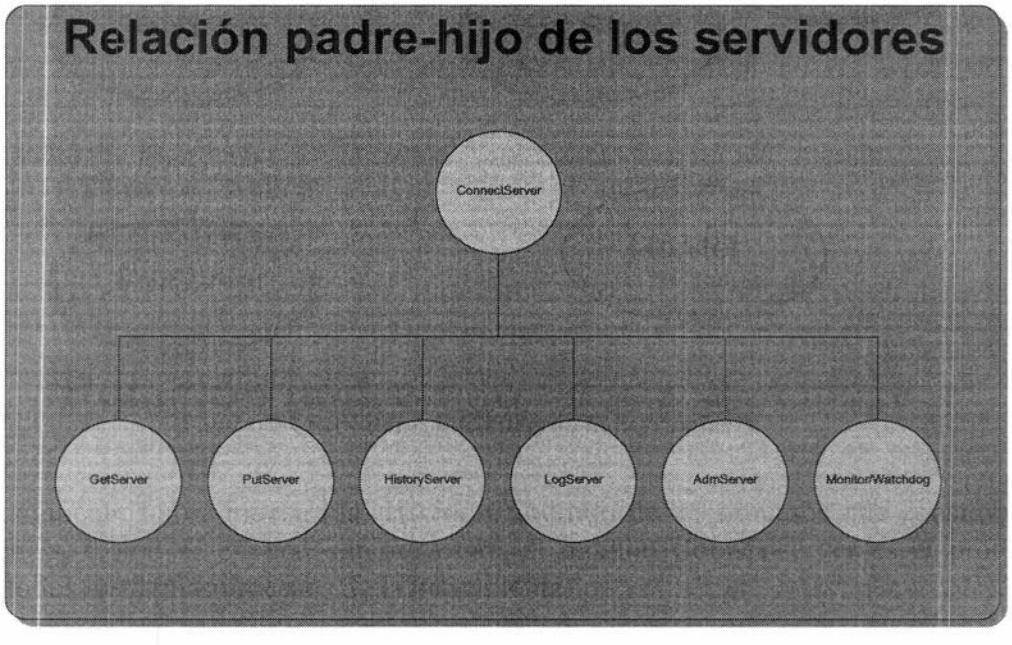


Figura 4 : Relación padre-hijo de los servidores

La siguiente figura muestra la jerarquía de servicios que existe entre los procesos que conforman los managers. Esta jerarquía muestra la relación cliente-servidor intrínseca a los procesos managers. En las secciones siguientes se estudiará en mayor profundidad que servicios proveen los diferentes módulos servidores a los otros procesos de la herramienta.

En esta figura no se muestra relación jerárquica con los clientes de los managers.

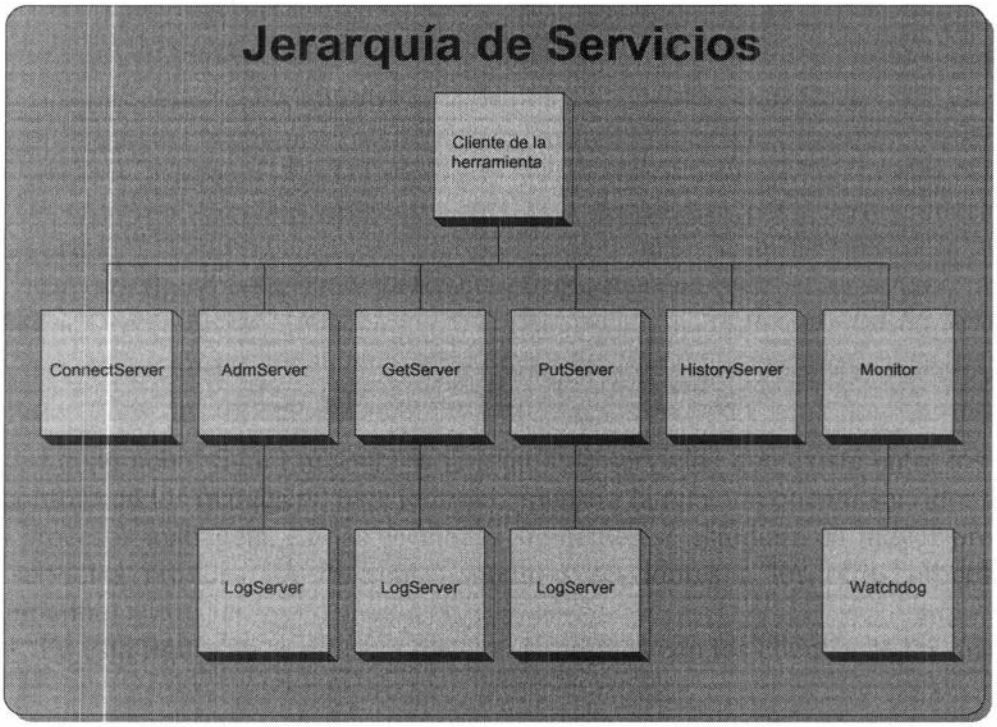


Figura 5 : Jerarquía de Servicios

5.1.2 Clientes de la herramienta.

Los clientes de los managers son aquellos procesos que utilizan las APIs provistas por los managers para manipular trabajos en un esquema de cooperación y flujo de trabajos.

Cuando un cliente de la herramienta comienza su ejecución debe seguir algunos pasos para que el manager lo reconozca como cliente y así poder utilizar las colas para manipular trabajos. Las operaciones provistas a los clientes son pocas y muy sencillas. Los pasos que debe realizar un cliente de los managers para utilizar las funciones provistas se explican a continuación.

1. Presentarse a los managers: mediante este paso se crea un nuevo identificador de cliente para los managers. El proceso que realiza la operación se convierte en un cliente y es monitoreado para notificar posibles caídas. Anta una falla se liberan sus trabajos pendientes.

   La función *Present* presenta un cliente a los managers e indica al servidor que realice un alta en su tabla de procesos activos. El tipo de proceso que utilizará el cliente indica las colas con las que se puede conectar y el tipo de operaciones que puede realizar sobre ellas. La función retorna un identificador único que el cliente utilizará en las llamadas a los manager.

2. Conectarse con cada una de las colas con las que interactua: Se debe conectar a todas las colas de entrada y/o salida. El servidor posee la información de seguridad que determina que colas puede acceder el cliente, cuales actúan como entrada y/o salida. Esta información está determinada por tipo de proceso con el que se presentó en cliente.

3. Utilizar las funciones de manipulación de trabajos: CrearToken, TomarToken, PasarToken, BorrarToken. Las mismas se estudiarán en el presente capítulo y tienen relación uno a uno con las operaciones descriptas en el capítulo 1, sección 1.2.1. Dependiendo del tipo de proceso del cliente existen restricciones a las operaciones que puede realizar.

   La función CrearToken crea nuevos trabajos introduciéndolos por primera vez en el flujo del sistema. TomarToken solicita un nuevo trabajo a los servidores. BorrarToken elimina un trabajo del flujo del sistema. PasarToken notifica que se terminó de procesar el trabajo y lo pasa de la cola origen a la destino. De esta manera habilita a los managers a entregar ese trabajo a algún proceso que lo solicite. BuscarToken toma un trabajo determinado de una cola.

Cada una de estas funciones, sus características y objetivos se explicarán en detalle durante el desarrollo del trabajo de tesis.

5.2 Diseño de las estructuras comunes y mecanismos de comunicación

En esta sección se introduce el mecanismo que utilizan los servidores que integran los managers para comunicarse y colaborar entre ellos. Debido a que cada servidor es un proceso separado, debe existir una estructura común a todos los servidores que almacene el estado de los trabajos, colas y procesos del sistema. Todos los servidores deben conocer el estado de los objetos que el sistema administra en forma sincronizada y consistente.

Se eligió como mecanismo para permitir la colaboración y comunicación de los servidores una integración de los mecanismos de IPC (Inter Process Communication). Se utiliza memoria compartida para compartir el estado de los objetos administrados, semáforos para sincronizar y proveer exclusión mutua en los accesos a memoria compartida y colas de mensajes para permitir a los servidores transmitir al servidor *logserver* la información de estado de trabajos a almacenar.

5.2.1 Diseño

Los objetivos de diseño son los siguientes:

- Proteger los recursos permitiendo la mayor generalidad y concurrencia posible.
- Obtener consistencia, unicidad y persistencia de los trabajos manejados por los managers.
- Proteger las estructuras de manera segura pero permitiendo buena performance.
- Lograr atomicidad de las operaciones que produzcan cambios en el estado.

5.2.1.1 Diseño de las estructuras de datos

Los objetos administrados por los managers y cuyo estado debe ser conocido por los servidores son la cola, el trabajo, el cliente y el tipo de proceso.

Se eligió almacenar cada uno de estos conceptos en tablas separadas en memoria compartida. Se utilizan encadenamientos entre los registros de las tablas para establecer la relación entre los objetos almacenados en las mismas. Se diseñaron las siguientes tablas:

- tabla de colas,
- tabla de trabajos,
- tabla de clientes,
- tabla de procesos.

Existe una única estructura, la tabla de trabajos, que contiene todos los trabajos administrados por el sistema, sin importar a qué cola pertenezcan o en qué estado se encuentren.

Mediante encadenamientos entre las tablas se almacena la relación entre los trabajos y las colas a las que pertenecen, así como el cliente que los está procesando. De esta manera, al cambiar el estado de un trabajo, ya sea porque pertenece a una nueva cola, o porque fue tomado o liberado por un cliente, solamente deben cambiarse los encadenamientos que correspondan. Se evita tener que mover datos de una estructura a otra. Como solamente se debe cambiar un encadenamiento al realizarse alguna operación sobre los trabajos, se mejora la performance evitándose el aumento en la complejidad de la administración de las tablas. Recordar que cada trabajo posee cierta información almacenada en la tabla de trabajos que debería ser copiada de una tabla de colas a otra si se implementara la alternativa de una tabla por cada cola.

Una alternativa a este diseño podría ser contar con una tabla por cada cola administrada por el sistema, debiéndose mover los trabajos entre tablas de colas. Se eligió la primer alternativa por mantener cada objeto en su propia tabla, permitiendo mayor claridad y por evitar el movimineto de trabajos entre tablas.

Con el esquema elegido puede pensarse lógicamente en cada cola por separado. Para conocer los trabajos que ella administra simplemente se siguen los encadenamientos. Para conocer los trabajos que un cliente está procesando, se siguen los encadenamientos desde la tabla de clientes. En el caso de las instancias del servidor *getserver* y *putserver*, cada vez que modifiquen un trabajo encolado en la tabla de trabajos, deberá también interactuar con las tablas de colas y de clientes, como consecuencia de los vínculos que se poseen y relacionan dichas tablas con la tabla de trabajos.

La siguiente figura muestra el conjunto de tablas diseñadas que residen en memoria compartida y cuyo objetivo es posibilitar la comunicación entre diferentes instancias de servidores concurrentes que componen a los managers. En la figura también se muestra con qué tablas interactúa cada servidor.

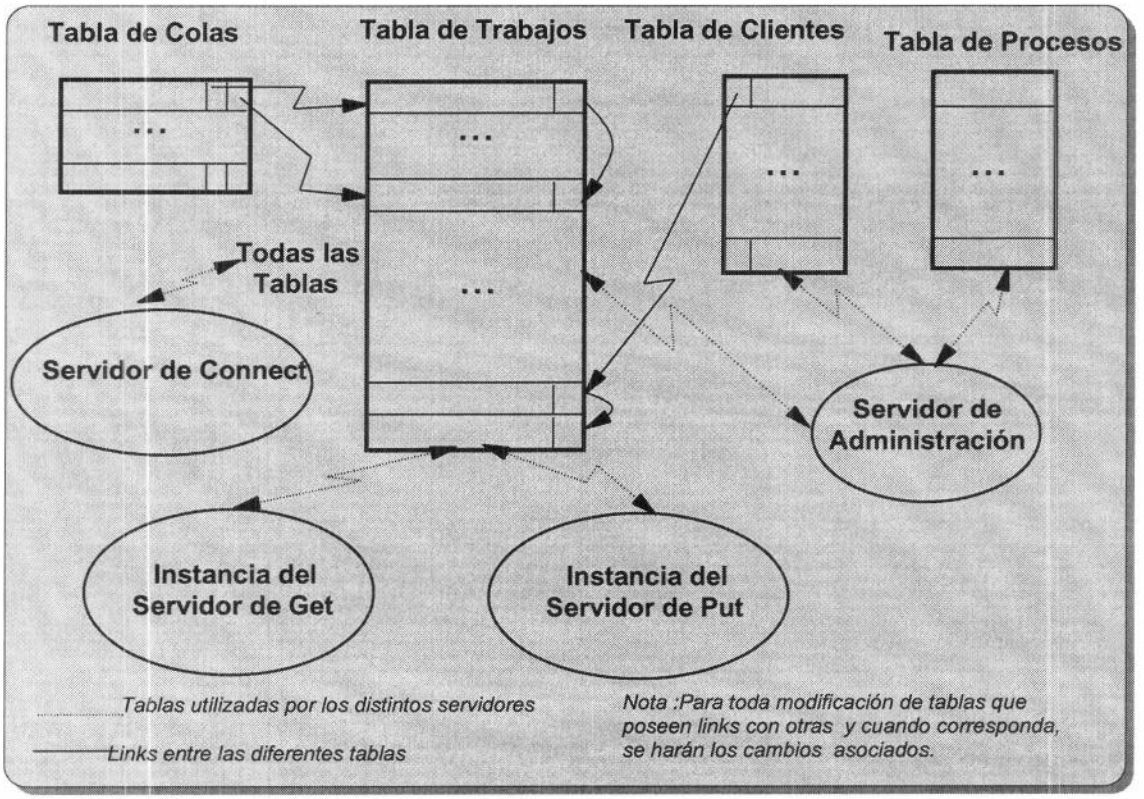


Figura 6 : Tablas de estructura de datos en memoria

Como se introdujo más arriba, el concepto de cola está representado mediante dos tablas. Una de ellas posee toda la información relacionada a las colas existentes en el sistema llamada ***Tabla de Colas*** La ***Tabla de Trabajos*** contiene los trabajos que se encuentran encolados. Para asociar un conjunto de trabajos a una cola, se diseñaron dos vínculos en cada registro de la cola, uno que apunta al primer trabajo de la misma, y otro que apunta al último. Para mantener el encadenamiento de los trabajos en una misma cola, en cada registro de la tabla de trabajos existe un vínculo hacia el siguiente trabajo en la misma cola.

En lo subsiguiente se utilizan datos que todavía no fueron detalladamente explicados. En esta sección sólo se introducen como integrantes de las tablas que se describen. Su funcionalidad y utilización serán explicados en detalle durante el Diseño de cada una de los servidores que los utilizan.

Tabla de Colas

La Tabla de Colas posee información asociada a cada cola definida en el sistema, datos como el nombre de la cola, el identificador dentro del sistema, la cantidad de trabajos encolados en la misma, un semáforo binario de exclusión mutua y los vínculos mencionados previamente.

El nombre de la cola es la forma que tiene un cliente de la herramienta para identificar la cola sobre la cual desea operar. El identificador es utilizado internamente para identificar las colas de una manera más eficiente que mediante el nombre. La cantidad de trabajos encolados se mantiene para poder implementar semáforos contadores infinitos y así independizarse de la limitación de la implementación de los semáforos de Unix. Existen dos vínculos hacia la tabla de Trabajos, uno hacia el primer trabajo de la cola y otro hacia el último trabajo. Estos existen por razones de eficiencia y para evitar recorrer todos los trabajos de la cola para encolar o para tomar el primer trabajo.

En la sección 6.3.1 "Implementación de las estructuras de datos", página 102 se detallan los campos de esta tabla y su función.

Tabla de Trabajos

En la tabla de trabajos se almacenan los datos de los trabajos que administra el sistema. Es el repositorio de la totalidad de los trabajos ubicados en las diferentes colas.

La tabla de trabajos contiene el nombre del trabajo, el identificador de la cola donde se halla el mismo y la cantidad de veces que el trabajo provocó la muerte del cliente que lo estaba procesando.

En la sección 6.3.1 "Implementación de las estructuras de datos", página 102 se detallan los campos de esta tabla y su función.

Tabla de Procesos

En esta tabla es donde se almacenan los diferentes Tipos de Procesos que el sistema soporta. Se mantiene la información de las colas que cada tipo de proceso puede acceder, a cuales tiene autorización para conectarse y que operaciones pueden realizar sobre las mismas. Esta tabla será consultada cada vez que un cliente se presente a los managers y cada vez que solicite conectarse a una cola.

En la sección 6.3.1 "Implementación de las estructuras de datos", página 102 se detallan los campos de esta tabla y su función.

Tabla de Clientes

Toda la información relacionada a los clientes que se encuentran utilizando los servicios provistos por los managers se almacena en esta tabla. Para agregar un nuevo cliente en la tabla es necesario conocer el Tipo de Proceso del mismo. Este dato es utilizado por la herramienta para verificar los permisos que posee sobre las diferentes colas del sistema, y el tipo de operaciones que puede realizar.

Pueden existir varias instancias de clientes con el mismo Tipo de Proceso, por lo tanto para poder identificarlas unívocamente, el cliente usará este identificador cuando solicite un servicio a los managers.

Cuando un cliente toma un trabajo, dicho trabajo pasa a un estado de retenido por el cliente en la cola. El cliente tiene un vínculo hacia la tabla de trabajos, manteniendo un encadenamiento de todos los trabajos que tiene retenidos y que está procesando.

En la sección 6.3.1 "Implementación de las estructuras de datos", página 102 se detallan los campos de esta tabla y su función.

5.2.1.2 Diseño del mecanismo de protección de recursos

Se debe diseñar e implementar un mecanismo de protección de los recursos compartidos para permitir la interacción de los servidores que integran los managers de manera sincronizada y sin interferencias.

Las estructuras descriptas en el punto anterior almacenan el estado de los objetos administrados por la herramienta y son accedidas y modificadas cuando un cliente realiza alguna operación. Deben protegerse de manera segura pero permitiendo buena performance.

A pesar que las modificaciones de las tablas son relativamente muy sencillas y se opera solamente sobre memoria, las operaciones deben ser atómicas ya que es fundamental operar concurrentemente sobre las tablas.

De no protegerse los recursos compartidos no sería posible acceder concurrentemente a la misma estructura sin crear problemas de consistencia. Como diferentes operaciones soportadas por los managers trabajan sobre distintos datos en las tablas se pretende lograr concurrencia entra aquellas operaciones que no operen directamente sobre los mismos datos. Este objetivo no es trivial debido a los encadenamientos que existen entre las diferentes tablas. Respondiendo a la naturaleza de las operaciones de mover trabajos entre colas, se pretende permitir acceso concurrente entre colas, para lograr la mayor performance y cantidad de operaciones procesadas por el sistema de administración de trabajos.

Por el diseño elegido para proteger a los recursos que explicaremos detalladamente en el siguiente punto, no se podrá realizar más de una operación sobre una cola dada en forma simultánea.

5.2.1.2.1 Utilización de semáforos para lograr exclusión mutua

Se utiliza un semáforo por cola para proteger a las mismas, donde cola se refiere al concepto de cola como almacenamiento de trabajos. En lo subsiguiente el término cola se refiere tanto a los trabajos libres para esa cola (los que aún no han sido tomados por ningún cliente), como los trabajos pertenecientes a la cola que están siendo procesados por clientes.

Dado el diseño de las tablas, una cola es la entrada de la misma en la tabla de colas, el encadenamiento de los trabajos libres pertenecientes a dicha cola que son almacenados en la tabla de trabajos y el encadenamiento de los trabajos apuntados desde la tabla de clientes de esa cola.

La estructura donde se almacenan los trabajos está organizada en registros o slots. La misma tiene slots vacíos que servirán para guardar los datos de los nuevos trabajos que se vayan creando en el sistema. El conjunto de slots libres de la estructura se maneja

como una cola más a la que llamaremos ***FreeSlot.*** Se mantiene un semáforo para proteger esta cola también.

En el siguiente punto se da una visión muy simplista y sencilla de los recursos que deben protegerse conceptualmente para cada una de las operaciones estudiadas. Sólo se describen los parámetros que influyen en el esquema de protección que deseamos mostrar.

Para cada una de las operaciones soportadas por los managers se presenta un pseudocódigo con los recursos que deben protegerse.

CrearToken(Cola, Cliente)

```
P(FreeSlot)          /* Exclusión mutua sobre la Lista de Slots libres */
P(SemCola)           /* Exclusión mutua para Cola en la que se creará el trabajo*/
< Actualiza encadenamiento de Slots Libres >
< Actualiza encadenamiento de NoTomados en Cola >
V(JobsCola)          /* Avisa que hay un trabajo más en la Cola */
V(FreeSlot)          /* Fin de exclusión mutua sobre FreeSlot */
V(SemCola)           /* Fin de exclusión mutua sobre Cola */
```

TomarToken(Cola, Cliente)

```
P(JobsCola)          /* Se fija si hay trabajos en esa cola, si no hay se bloquea */
P(SemCola)           /* Pide exclusión mutua sobre la Cola */
< Actualiza encadenamiento de NoTomados en Cola >
< Actualiza encadenamiento de Tomado Cliente en Cola >
V(SemCola)           /* Fin de exclusión mutua sobre Cola */
```

BuscarToken(Cola, Cliente)

```
/* Si hay trabajos disponibles en esa Cola */
P(SemCola)           /* Pide exclusión mutua sobre la Cola */
< Actualiza encadenamiento de NoTomados en Cola >
< Actualiza encadenamiento de Tomado Cliente en Cola >
V(SemCola)           /* Fin de exclusión mutua sobre Cola */
```

TomarToken y BuscarToken se comportan de igual manera en lo referido a protección de los recursos, cuando hay trabajos en la Cola.

PasarToken(Cola1, Cola2, Cliente)

```
P(SemCola1)          /* Pide exclusión mutua sobre la Cola1 */
P(SemCola2)          /* Pide exclusión mutua sobre la Cola2 */
< Actualiza encadenamiento de Tomado Cliente en Cola1 >
< Actualiza encadenamiento de NoTomados en Cola2 >
V(JobsCola2)         /* Avisa que hay un trabajo más en la Cola2 */
V(SemCola1)          /* Fin de exclusión mutua sobre Cola1 */
V(SemCola2)          /* Fin de exclusión mutua sobre Cola2 */
```

En este caso no es necesario realizar un *P(JobsCola1)* indicando que se tomó (disminuyó la cantidad de trabajos en la Cola1) un trabajo en Cola1 ya que esto se realizó durante la operación TomarToken que siempre es previa a PasarToken.

BorrarToken(Cola, Cliente)

```
P(FreeSlot)          /* Pide exclusión mutua sobre la Lista de Slots libres */
P(SemCola)           /* Pide exclusión mutua sobre la Cola */
< Actualiza encadenamiento de Slots Libres >
< Actualiza encadenamiento de Tomado Cliente en Cola >
V(FreeSlot)
V(SemCola)           /* Fin de exclusión mutua sobre Cola */
```

ClientDown(Client)

```
P(ColaRechazados)    /* Pide exclusión mutua sobre la Cola de Rechazados */
P(ColasIn)           /* Pide exclusión mutua sobre las Colas de Entrada */
< Actualiza encadenamiento de Trabajos Tomados por Cliente en las Colas >
V(ColasIn)           /* Fin exclusión mutua sobre las Colas de Entrada */
V(ColaRechazados)    /* Fin exclusión mutua sobre la Cola de Rechazados */
```

5.2.1.2.2 Consideraciones relativas al mecanismo de protección

Dado el esquema de tablas de memoria donde se almacenan los datos y los encadenamientos de los mismos, para que el sistema de protección sea completamente general a cualquier arquitectura hay que bloquear todas las colas de entrada del tipo de proceso del cliente que realiza una operación. De esta manera se garantiza que no haya accesos que violen la consistencia de los encadenamientos del cliente. Basta con bloquear con el PColas todos las colas de entrada para ese tipo de proceso. Sólo es necesario recorrer una tabla en memoria para saber cuales son las colas de entrada para el proceso.

Si sólo se tiene una cola de entrada por tipo de proceso, el sistema solamente bloqueará esa cola de entrada.

Para lograr completa concurrencia, en BorrarToken se debería bloquear sólo al cliente y no a las colas del mismo porque solamente se agrega a la lista de slots libres, no a la de los trabajos libres de esa cola. Esta no es una operación que ocurra muy frecuentemente.

Hay que tener en cuenta el orden en que se reservan los recursos usando los semáforos para evitar situaciones de deadlock. El problema potencial es el siguiente:

El proceso A realiza la siguiente operación pasando un trabajo de Cola1 a Cola2

PasarToken(Cola1, Cola2, Cliente)
P(SemCola1) (1)
P(SemCola2) (2)
< Actualiza encadenamiento de Tomado Cliente en Cola1 >
< Actualiza encadenamiento de NoTomados en Cola2 >
V(JobsCola2)
V(SemCola1)
V(SemCola2)

El proceso B realiza la misma operación pero pasando un token de Cola2 a Cola1

PasarToken(Cola2, Cola1, Cliente)
P(SemCola2) (3)
P(SemCola1) (4)
< Actualiza encadenamiento de Tomado Cliente en Cola1 >
< Actualiza encadenamiento de NoTomados en Cola2 >
V(JobsCola2)
V(SemCola1)
V(SemCola2)

Si se diera la secuencia de ejecución (1), (3) se produciría deadlock ya que ambos procesos A y B se bloquearían al realizar (2) y (4) y nadie liberaría los recursos.

Para la evitar este problema se utilizaron arrays de semáforos de UNIX para lograr atomicidad en la entrada y salida de las zonas críticas. El objetivo es evitar la reserva de recursos. Debido a que se usa siempre el mismo conjunto de semáforos, la posible desventaja podría ser que se está secuencializando todas las entradas y salidas a las zonas críticas en un único semáforo.

Para más detalles referirse al Anexo B: Semáforos de UNIX.

5.2.1.2.3 Orden de atención de las solicitudes

Cada servidor que desee acceder a los recursos cuando no estén disponibles se encolará al ejecutar P (operación *semop* de UNIX) del semáforo.

La definición conceptual de semáforo no especifica que se deba encolar en un determinado orden sino solamente que se bloqueen los procesos hasta tanto no tengan disponible el recurso solicitado. Las herramientas para implementar la idea conceptual de semáforos deben manejar a los bloqueados y los encolan. En el caso de UNIX es un orden fifo (first in, first out) relativo a los pedidos en condiciones de ser atendidos.

5.3 Diseño del servidor de Connect: connectserver

El *connectserver* es el padre de todos los otros servidores: *getserver, putserver, administrador, historyserver, logserver y monitor*. Crea los procesos que luego conformarán la estructura de los *managers*.

Es el encargado de inicializar todas las estructuras comunes a los servidores como ser tablas de memoria compartida, semáforos y colas de mensajes.

Es un servidor de RPC que se presenta para recibir las solicitudes que realizan los clientes para conectarse a las colas.

5.3.1 Diseño

A partir de la información proporcionada por el esquema de colas, se calcula la cantidad de instancias de cada uno de los servidores a crear, así como el tamaño de las estructuras para almacenar los trabajos. A cada arquitectura se le asocia la configuración más adecuada para su ejecución. Esto se realiza en tiempo de ejecución, sin necesidad de modificar o recompilar el código de los managers, lográndose así uno de los objetivos del trabajo.

Luego de la inicialización de los servidores, el *connectserver* se transforma en el punto de entrada para los clientes. Estos solicitan al connectserver la conexión para las colas con las que operarán.

Antes de presentarse, mediante la operación *Present*, los clientes solamente conocen la dirección de RPC del connectserver. Al presentarse y conectarse a las colas, se le proporciona al cliente la información necesaria para que pueda acceder a los servicios de los demás servidores. Se otorga la identificación de RPC de cada una de las instancias de los servidores que sirven a las colas con las que se conectó. Solamente se otorga información que le compete a ese proceso, dependiendo del tipo de proceso que declaró al ejecutar el *Present* y de las colas con las que se conectó. Se aplica seguridad a nivel de conexión.

5.3.1.1 Estructuras de datos.

Las estructuras de datos utilizadas por el connectserver son las estructuras detalladas en la sección 5.2.1.1 "Diseño de las estructuras de datos", página 46 y otras estructuras privadas del mismo.

El connectserver crea todas las estructuras de datos y las inicializa. También exporta los identificadores de las tablas para que puedan ser accedidas por los demás servidores que integran los managers, permitiendo la cooperación entre los mismos mediante el uso de recursos compartidos.

5.3.1.2 Estructura del connectserver.

El connectserver es un servidor de RPC.

El punto de entrada para el mismo es la función Inicializar. Esta función se ejecuta cuando se comienza este servidor y es la encargada de inicializar las estructuras, setear las acciones a tomar en caso de terminación normal o anormal del servidor y crear todas las instancias de los procesos servidores.

Una vez ejecutada la función Inicializar, el connectserver se transforma en un servidor cuyo único servicio es el prestado a los clientes de los managers a través de la función ConectarCola. Esta función permite a los clientes conectarse a las colas con las cuales operará.

5.3.1.3 Acciones realizadas por el connectserver.

En la presente sección se presentan las acciones realizadas por cada una de la principales funciones del connectserver.

La función Inicializar se ejecuta antes que el connectserver se transforme en servidor de RPC. Esta función es la responsable de crear el ambiente de ejecución de los restantes procesos integrantes de los managers, así como inicializar las estructuras de datos.

1) Inicializar

1. Realiza las acciones necesarias para que el connectserver sea un demonio de UNIX y se asegura que exista una única instancia del servidor ejecutando.
2. Setea las señales para especificar las acciones a tomar en caso de muerte normal o anormal del connectserver y de cualquiera de los servidores hijos del mismo.
3. Setea el umask para establecer los permisos de ejecución correctos.
4. Inicializa el syslog.
5. Lee la configuración para el esquema actual.
6. Inicializa las tablas del sistema. Estas son las estructuras de memoria compartida utilizadas por todos los servidores.
7. Inicializa los semáforos compartidos por todos los servidores.
8. Crea las instancias de cada uno de los servidores para el esquema actual.
9. Almacena los nombres de los recursos del sistema que están siendo utilizados por la ejecución actual de los managers.

Una vez que el connectserver exporta la función ConectarCola, se ha transformado en servidor de RPC. Los clientes de la herramienta solamente pueden acceder a esta función del Connectserver. Al ser un servidor de RPC, el connectserver ejecuta un ciclo

infinito esperando el pedido de servicio que los clientes solicitan a través de ConectarCola.

2) ConectarCola

Es la función encargada de manejar la seguridad. Cualquier requerimiento asociado a control se desarrollará en este módulo.

1. Verifica la validez del identificador del cliente.
2. De acuerdo a los permisos de acceso del cliente retorna los identificadores de la cola adecuados para que el mismo opere.

5.4 Diseño del servidor de get: getserver

Este servidor es el encargado de recibir los pedidos de trabajos que realizan los clientes que usan las APIs de la herramienta.

Este módulo tiene la función de entregar trabajos a los clientes que así lo requieran. Brinda dos clases de servicios, uno bloqueante y otro no bloqueante.

Controla la correcta administración de los trabajos asegurando la consistencia en el flujo de los mismos entre las colas y los pedidos de los clientes.

5.4.1 Diseño

Se garantiza la unicidad en la entrega de los trabajos, no puede haber dos clientes procesando el mismo trabajo.

Con el objetivo de ser liberados en caso de la caída del cliente, se provee la identificación del trabajo o los trabajos que están siendo procesados por un determinado cliente.

El orden de entrega de los trabajos a los clientes es de fundamental importancia para la semántica asociada al esquema administrado. No entra dentro del alcance del presente trabajo de tesis el estudio e implementación de un esquema de ordenamiento y prioridades de los trabajos. Por lo tanto, la adminsitración de la cola se definió en un módulo independiente y es fácilmente ampliable a soportar cualquier otro esquema de ordenamiento o prioridades para la entrega de trabajos. El getserver está implementado para administrar una cola FIFO, el primer trabajo que se deposita es el primer trabajo en ser entregado ante un requerimiento. Una variedad de servicio provisto es la entrega de un trabajo específico solicitado por un cliente.

El getserver provee servicios bloqueante y no bloqueante de entrega de trabajos a los clientes.

Requerimientos que no pueden satisfacerse. Servicio bloqueante.

Debe diseñarse el comportamiento del getserver para todos los requerimientos de los clientes. Incluyéndose los casos normales de operación, que no constituyen un error, pero para los cuales los managers no pueden satisfacer los requerimientos del cliente.

Un caso normal y que no consiste en un error se da cuando el getserver no puede cumplir un pedido de trabajo porque no existen más trabajos disponibles.

La semántica del cliente es la que determina la acción adecuada del mismo frente a esta circunstancia. Esta fuera del alcance de la herramienta determinar cual sería la acción más indicada para cada tipo de cliente. En la mayoría de los casos, los clientes que se benefician de una arquitectura como las representadas, se comporta como un ciclo infinito tomando trabajos, procesando y entregando trabajos para que sean procesados por otros clientes. Esto indica que ante la falta de trabajos disponibles, no podría seguir ejecutando

porque no tiene otra cosa útil que ejecutar. Ante este perfil de procesamiento, no tiene mucho sentido devolverle el control al cliente por las siguientes dos razones:

1. El cliente debería manejar su propio busy-waiting consumiendo recursos;
2. El servidor tampoco tiene procesamiento para realizar si no hay trabajos que administrar ni entregar.

Decidimos que el servidor se bloquee en espera del arribo de trabajos para la cola que él administra. No le devolverá el control al cliente hasta que esté en condiciones de satisfacer su requerimiento. Esta razón de diseño es la razón principal por la cual se decidió separar el módulo de entrega de trabajos, del módulo que acepta trabajos para encolar. El bloquear al getserver no debe impedir la aceptación de nuevos trabajos.

Requerimientos de trabajos específicos. Servicio no bloqueante.

El sistema de procesamiento bloqueante es válido para la mayor parte de los clientes para los cuáles está definida esta arquitectura, pero no para su totalidad.

El segundo escenario presenta a un cliente que desee pedir un trabajo especifico identificándolo. De encontrarse disponible este trabajo en la cola especificada, le será entregado. Por este motivo el diseño debe soportar una versión no bloqueante del pedido de trabajos, ya que si un trabajo específico no se encuentra, no se puede retener al cliente, ni tampoco al servidor. El que no exista el trabajo requerido no implica que no haya más trabajos para entregar.

Se asocia una instancia de servidor getserver a cada cola que exista en el esquema. Cuando una instancia del getserver se bloquea por no haber más trabajos en la cola que administra, las demás instancias de getserver continúan con su accionar normal. Se permite una administración centralizada y consistente de los trabajos a través de las estructuras de datos compartidas.

El getserver brinda , de acuerdo a las premisas básicas:

- Servicio bloqueante si no hay trabajos para entregar.
- Servicio no-bloqueante ante pedidos de trabajos particulares que no pueden ser satisfechos.
- Consistencia de la información.

5.4.1.1 Estructura de datos.

Las estructuras de datos utilizadas por el getserver son las estructuras detalladas en la sección 5.2.1.1 "Diseño de las estructuras de datos", página 46 y otras estructuras privadas del mismo.

El getserver utiliza las estructuras de datos que almacenan información relativa a clientes, trabajos, colas y tipos de procesos.

5.4.1.2 Estructura del getserver.

El getserver es un servidor de RPC.

El punto de entrada para el mismo es la función ParseArgs. Esta función se ejecuta cuando comienza este servidor y se encarga de parsear los argumentos que le pasa el connectserver al getserver en el momento de realizar el fork del mismo. El getserver obtiene los identificadores de las tablas en memoria compartida y del conjunto de semáforos que utilizará a través de estos parámetros. Esta función inicia el logging en el syslog.

Una vez ejecutada la función ParseArgs, el getserver se transforma en un servidor que provee los servicios implementados por las funciones TomarToken y BuscarToken.

5.4.1.3 Acciones realizadas por el getserver.

Este módulo consta de una función de inicialización y dos funciones principales. Las funciones principales se diseñaron para soportar el servicio bloqueante y no bloqueante que este servidor brinda a los clientes de la herramienta.

Servicio bloqueante.

La función asociada al servicio bloqueante es TomarToken que recibe como parámetros la identificación del cliente que está haciendo el requerimiento y el identificador de la cola de donde se desea tomar el trabajo. El valor retornado es el identificador del trabajo. Si no se puede satisfacer el requerimiento, el servidor se bloquea, bloqueando en consecuencia al cliente hasta que sean depositados nuevos trabajos en esa cola.

El servidor se bloqueará siempre que no haya nada que tomar. En caso contrario entrega el primer trabajo de la cola. Se debe conseguir la exclusión mutua para acceder a todas aquellas estructuras compartidas entre las diferentes instancias de servidores que componen el sistema. El trabajo tomado se asocia a la lista de trabajos del cliente.

Para soportar el comportamiento bloqueante se diseñó un esquema de semáforos asociados al acceso a todo recurso compartido que lo necesitase. Para mayores detalles relativos al manejo de los semáforos referirse al Anexo A.

Servicio no bloqueante.

Para la modelización del servicio no bloqueante se diseñó la función BuscarToken que recibe la misma información que TomarToken más el nombre del trabajo que se

requiere. Esta función retorna un código indicando si tuvo éxito la búsqueda del trabajo o no. Si la función recibe como nombre del trabajo requerido el valor nulo, retorna el primer trabajo disponible.

Esta función no se bloquea en un semáforo en caso de no exitir trabajos. Se consulta la variable asociada a la longitud de la cola de trabajos encolados en la cola pedida, si este valor es mayor a cero, se solicita exclusión mutua para el acceso a los recursos compartidos y se procede a buscar el trabajo pedido por el cliente. Si la búsqueda no es exitosa, se retorna un código de error que refleje este resultado. Sino, se procede de la misma manera que en la función TomarToken antes descripta.

Inicialización.

La tercer función de este servidor de trabajos, se encarga de la inicialización del entorno para la correcta ejecución de este servidor en sincronización con los demás servidores de la herramienta.

5.5 Diseño del servidor de put: putserver

Este servidor es el encargado de recibir los pedidos para agregar nuevos trabajos a la cola que está administrando. Las funciones soportadas por este módulo no son bloqueantes. No existen servicios que mantengan bloqueado tanto al cliente como al servidor, exceptuando los bloqueos temporales consecuencia de esquemas de acceso a regiones críticas.

El putserver brinda toda clase de servicios relacionados a la manipulación de trabajos excluyendo la entrega de los mismos, que es responsabilidad del getserver. Las funciones de este servidor proveen los servicios de agregar un trabajo a una determinada cola una vez que el cliente terminó de transformarlo. También permite poner un trabajo en una cola inmediatamente después de haber sido creado, introduciendo un trabajo nuevo al flujo del sistema. Finalmente, permite borrar un trabajo, eliminándolo del flujo total.

Este servidor, como los demás que integran la herraminta, garantiza la consistencia de las operaciones.

5.5.1 Diseño

Todas las operaciones que realice este módulo se traducen en un cambio del estado del trabajo. Como las estructuras de datos que almacenan el estado de colas, trabajos, clientes y tipos de procesos son tablas en memoria compartida, una falla implicaría perder el estado general en que se encontraba el sistema. Por este motivo, se decidió utilizar un mecanismo de logging en combinación con un mecanismo de manejo de historia, los cuales se explican con más detalle en las secciones de servidor de logging y servidor de historia. Se registra cada movimiento o cambio de estado, asociados a operaciones del putserver, permitiendo llevar una imagen del estado del sistema y posibilitar la reconstrucción de las colas en caso de falla.

En el caso del getserver, como un trabajo se entrega a un cliente, el trabajo está retenido, pero no cambia de cola. No es necesario registrar el estado de retenido, ya que de fallar el servidor, o el cliente, al no haberse registrado cambio alguno en el estado del trabajo , éste se reincorporará a la cola que lo albergaba. Esto provocará a lo sumo, que se procese el mismo trabajo más de una vez.

A diferencia del getserver, las tres operaciones soportadas por este módulo deben registrar todas sus acciones, ya que cada una de ellas introduce, o borra o cambia de cola un trabajo modificando el estado de las colas. La tarea de registro consiste en llevar un logging de movimientos.

Las operaciones que brinda el putserver responden a la necesidad de otorgar a los clientes de la herramienta los servicios para cumplir las siguientes funciones:

Creación de nuevos trabajos.

Se dice que un trabajo se "crea" o ingresa al flujo de datos del sistema, cuando el trabajo no existe previamente y no es producto de ninguna transformación. Un trabajo se crea cuando un cliente, que no tenía ningún trabajo retenido para transformar, genera con sus propios datos un nuevo trabajo y lo ingresa al sistema agregándolo en alguna de las colas.

Un trabajo mantiene su "identidad" a lo largo de todo el proceso de transformaciones. Físicamente, el trabajo puede cambiar, debido a las transformaciones sucesivas que los procesos clientes efectúan sobre el mismo, pero mantiene su identificación.

Borrado de trabajos.

Cuando un cliente ejecuta un TomarToken (operación descripta en la sección referida al getserver) ocasiona que el sistema registre la existencia del trabajo como siendo procesado por el cliente. Esto pasa el trabajo a un estado de *retenido* sin ser eliminado realmente de la cola de donde fue tomado. El cliente puede transformar el trabajo y obtener el resultado final deseado.

Se dice que un trabajo se "borra" cuando el cliente que lo tiene retenido solicita la eliminación del trabajo del flujo del sistema. Al ser borrado, se elimina de la cola donde se encontraba retenido.

La semántica más generalmente asociada a la operación borrar es la de compleción de proceso del trabajo.

Cambio de cola de un trabajo.

Cuando un cliente ejecuta un TomarToken (operación descripta en la sección referida al getserver) ocasiona que el sistema registre la existencia del trabajo como siendo procesado por el cliente. Esto pasa el trabajo a un estado de *retenido* sin ser eliminado realmente de la cola de donde fue tomado. El cliente puede transformar el trabajo y obtener el resultado final deseado. Si el trabajo no ha sufrido todas las transformaciones debidas, el cliente debe colocarlo en la cola destinada a la próxima etapa de transformación del mismo.

Se dice que un trabajo se *pasa* de una cola a otra, cuando se toma un trabajo para transformarlo, sin ser ésta la última etapa transformación. Luego de procesarlo, se pasa el trabajo de la cola donde se encontraba retenido a una nueva cola para así continuar con el

esquema de transformaciones asociado. El pasaje de una cola a la siguiente debe ser una operación atómica para soportar la persistencia de los trabajos.

Resumimos las características mencionadas:

- Servidor no bloqueante.
- Crear implica agregar un nuevo trabajo en una cola.
- Borrar implica eliminar un trabajo retenido en una cola por un determinado cliente.
- Pasar implica eliminar un trabajo retenido en una cola por un determinado cliente para agregarlo en la cola destino. Este pasaje debe constituir una operación atómica.

5.5.1.1 Estructura de datos.

Las estructuras de datos utilizadas por el putserver son las estructuras detalladas en la sección 5.2.1.1 "Diseño de las estructuras de datos", página 46 y otras estructuras privadas del mismo.

El putserver utiliza las estructuras de datos que almacenan información relativa a clientes, trabajos, colas y tipos de procesos.

5.5.1.2 Estructura del putserver.

El putserver es un servidor de RPC.

El punto de entrada para el mismo es la función ParseArgs. Esta función se ejecuta cuando comienza este servidor y se encarga de parsear los argumentos que le pasa el connectserver al putserver en el momento de realizar el fork del mismo. El putserver obtiene los identificadores de las tablas en memoria compartida y del conjunto de semáforos que utilizará a través de estos parámetros. Esta función inicia el logging en el syslog.

Una vez ejecutada la función ParseArgs, el putserver se transforma en un servidor que provee los servicios implementados por las funciones CrearToken, BorrarToken y PasarToken.

5.5.1.3 Acciones realizadas por el putserver.

Este módulo consta de una función de inicialización y tres funciones principales.

Creación de trabajos.

La función asociada al servicio de crear un trabajo es CrearToken que recibe como parámetro la cola donde se debe encolar el nuevo trabajo y la identificación del mismo.

Esta operación se compone de un conjunto de operaciones de inserción de un trabajo en la cola especificada, respetando el esquema de administración de las colas definidas. Se actualizan todas las variables asociadas a la cola para reflejar que se ha introducido un nuevo trabajo al flujo del sistema. El acceso a las estructuras de datos compartidas por los diferentes servidores es manejado a través del mismo esquema de semáforos utilizado por el getserver. Se retorna un valor indicando si la operación fue exitosa o no.

Borrado de trabajos.

La función asociada al servicio de borrar un trabajo es BorrarToken. Recibe como parámetros el identificador del cliente que tenía tomado el trabajo, el identificador del trabajo que se desea borrar y la cola donde se encuentra retenido dicho trabajo. Un cliente puede tener tomados varios trabajos concurrentemente.

El servidor modifica el estado de retenido sin asociarle ningún otro estado ni encolarlo en ningún otra cola. Para hacerlo, debe tener derecho de acceso a las estructuras en forma exclusiva, garantizando la consistencia de las operaciones. No se permite borrar un trabajo que no esté retenido por un cliente, es decir, borrarlo de una cola sin procesarlo.

Cambio de cola de un trabajo.

La función asociada al servicio de pasar un trabajo de una cola a otra es PasarToken. Recibe el identificador del cliente que tenía tomado el trabajo, el nombre de la cola donde se encuentra retenido el trabajo, el nombre de la cola a donde se va a agregar, y el identificador del trabajo. Este último parámetro se agrega por razones de validación debido a que un cliente puede tener más de un trabajo retenido.

Los mecanismos para acceder a la región crítica y poder modificar los datos comunes son más complicados debido a que hay que acceder simultáneamente a la información de dos colas, y a la información del cliente. Una vez que se obtiene exclusión mutua, se procede a *pasar* el trabajo de una cola a la otra.

Esta función es similar a la composición de las dos funciones previamente descriptas, BorrarToken y CrearToken. No se puede pasar un trabajo que no esté retenido por un cliente. La razón es que se pasa un trabajo una vez de haberle aplicado algún tipo de transformación, si directamente se pasa de una cola a otra significa que no había necesidad de haber pasado por esa cola para llegar al resultado definitivo. La razón por la cual se introduce una operación nueva en lugar de utilizar las operaciones de BorrarToken y CrearToken, radica en la necesidad de que esta operación sea atómica. Si se ejecutara como dos sentencias separadas podría ocurrir una falla entre la ejecución de las mismas provocando la pérdida o inconsistencia del estado de los trabajos.

Las operaciones BorrarToken y PasarToken que involucran un cliente que tenía retenido un trabajo, ejecutan rutinas de validación del cliente.

Inicialización.

La cuarta función de este módulo, se relaciona con la inicialización del entorno para la correcta ejecución de este servidor en sincronización con los demás componentes que conforman la herramienta.

5.6 Diseño del servidor de logging: logserver

El objetivo de este servidor es proveer un mecanismo seguro para almacenar en un medio persistente todas las operaciones significativas realizadas por los servidores que integran los managers.

Es de fundamental importancia el cuidado de la performance de este servidor debido a que es un cuello de botella. Bloquea el accionar de todos los servidores hasta tanto no se haya efectuado la escritura en medio persistente de las operaciones.

5.6.1 Diseño.

El servidor de logging actúa como servidor para todos los demás procesos que conforman los managers. Cada uno de los servidores es un cliente para el servicio que presta el logserver.

Para asegurar la persistencia de las operaciones ante una caída, cada servidor debe enviar un mensaje al servidor de logging registrando la operación realizada. Esta llamada es bloqueante, y el invocante recién recupera el control una vez que el servidor de logging ha realizado la escritura en almacenamiento persistente. De ahí la importancia de poseer una muy buena performance para el accionar de este servidor.

Introduciremos en una primera parte los conceptos de logging que deseamos resaltar en este diseño. Se han estudiado diversos esquemas utilizados y en investigación actualmente, como los explicados en Daniels [027], Wang [025], Ruffin [022] y Silva [026]. Cada uno de estos autores propone o estudia diferentes técnicas de logging en diferentes ambientes.

En la siguiente sección se considera un servicio de log a nivel sistema operativo como el planteado en Daniels [027] en contraposición al servicio de log provisto por los managers que se estudiará en la sección 5.5.3 "Diseño del mecanismo de logging para los managers".

5.6.2 Conceptos de logging aplicados en el diseño. Log de sistema operativo versus aplicación.

Existen varias posturas referida a quien debería proveer los servicios de logging. En nuestro caso, por las razones que se explicarán durante el diseño del servidor de logging hemos decidido que los mismos managers provean el mecanismo de logging para asegurar persistencia de la información manejada. En esta sección se mostrará la postura de algunos investigadores como Daniels [027] que respaldan la idea de que estos servicios sean provistos por el sistema operativo.

Según lo que manifiestan los investigadores del Almaden Research Center [025], los log para recuperación (recovery logs) son un ítem importante en la computación

distribuida que se debería proveer como parte del sistema operativo señalan que una adaptación trivial del manejo de logs en las bases de datos tradicionales no es adecuado como servicio de logging compartido por múltiples administradores de recursos.

Los logs para recuperación son importantes para la computación con tolerancia a fallas porque son usados por coordinadores de transacciones y administradores de recursos transaccionales, para registrar en forma confiable la información necesaria para implementar la atomicidad, seriabilidad, y propiedades de permanencia de las transacciones.

Las transacciones y las técnicas de recuperación basadas en logs relacionadas, simplifican la programación tolerante a fallas, garantizando la atomicidad de los programas que ejecutan en múltiples nodos y haciendo permanente los datos aún en presencia de fallas en el nodo. Adicionalmente, las técnicas de recuperación basadas en logs permiten un recupero parcial ante caídas catastróficas con destrucción de todas las copias de los datos. Las transacciones facilitan la tolerancia a fallas garantizando que, o bien todas las acciones de un programa distribuido (transacción) ocurren y se hacen visibles fuera de esa transacción, o bien, todos sus efectos se deshacen. De esta manera los programadores de aplicaciones distribuidas tolerantes a fallas que utilizan recursos transaccionales, no escriben código especial para deshacer los efectos de una falla parcial. En forma similar, la propiedad de seriabilidad de las transacciones libera a los programadores de la preocupación surgida de la ejecución concurrente de sus programas.

Algunas técnicas de tolerancia a fallas como la replicación se complementan con transacciones atómicas. Las transacciones garantizan invariantes de la replicación. En el evento de una falla, una transacción puede recomenzarse usando una copia diferente de los mismos datos. Transacciones anidadas facilitan la construcción de sistemas tolerantes a fallas usando replicación por medio de contener las fallas, permitiendo a las transacciones continuar usando recursos redundantes.

La robustez es una ventaja de los mecanismos de recuperación basados en logs. A diferencia de otras técnicas de tolerancia a fallas que replican datos en memoria volátil en múltiple nodos, los logs son una forma de almacenamiento estable que sobrevive la falla completa temporal de todos los nodos del sistema distribuido. Pueden extenderse a técnicas de recuperación ante desastre (disaster recovery) desde archivos de logs, cuando los datos en todos los almacenamientos online ha sido destruida. La recuperación ante desastre restablece información consistente, pero tal vez no actualizada en el sistema.

La idea de transacciones y la recuperación basada en logs se originaron con los sistemas de gestión de base de datos. A medida que las transacciones y la recuperación basada en logs se usa en una más diversa gama de aplicaciones con tolerancia a fallas, se explora la integración de la administración de transacciones y logging de recuperación con los sistemas operativos. Estas facilidades de transacción distribuida (Distributed Transaction Facilities - DTF) simplifican la construcción y mejoran la performance de las aplicaciones tolerantes a fallas que acceden a múltiples recursos transaccionales.

Un facilidad de transacción distribuida (DTF) aumenta un sistema operativo con administración común de transacciones y facilidad de recuperación. Los administradores de recursos usan las facilidades compartidas de los DTF (así como RPC y otras facilidades programación distribuida que brinda el sistema operativo) para implementar objetos transaccionales. Ejemplos de administradores de recursos transaccionales pueden ser sistemas de archivos, administradores de bases de datos, object stores, y monitores de teleprocesamiento. Cada administrador de recursos puede desarrollarse y adminstrarse en forma independiente y pueden ubicarse sobre otros administradores de recursos en un modelo de capas (por ejemplo, un DBMS puede usar un sistema de archivos).

Las aplicaciones tolerantes a fallas componen operaciones sobre múltiples recursos en transacciones (que pueden ser automáticamente recomenzadas si una falla temporal aborta una transacción o subtransacción). Los diseñadores de los administradores de recursos no necesitan anticipar todas las diferentes combinaciones de maneras en que las aplicaciones compondrán operaciones ya que el mecanismo de transacciones subyacente puede garantizar consistencia entre administradores de recursos.

La administración de transacciones y los servicios de logging ofrecen mejoras en la performance comparados con facilidades implementadas sobre administradores de recursos individuales. Los servicios de logging mejoran la performance de las transacciones distribuidas así como las locales que acceden a múltiples administradores de recursos. La coordinación de los commit de las transacciones disminuye el número de mensajes que deben enviarse entre nodos distribuidos para hacer commit de una transacción distribuida. Las transacciones locales son más rápidas porque una escritura a un almacenamiento persistente puede forzar a los registros de log de todos los administradores de recursos junto con el registro sobre el que se hace commit de la transacción.

Lógicamente, los log de recuperación de las bases de datos son secuencias append-only de registros sobre almacenamiento persistente en los que hay tres operaciones importantes. La operación de escritura toma un registro de log como argumento, lo bufferiza en memoria volátil y retorna un Número de Secuencia de Log (LSN) que se usa en lo posterior para identificar al registro de log. La operación de "force" toma un LSN como argumento y retorna cuando todos los registros con LSN menores o iguales al argumento han sido escritos en un almacenamiento persistente. La operación de lectura retorna el registro de log asociado con el LSN pasado como argumento. Una implementación eficiente de estas tres operaciones es esencial para un servicio de log compartido.

Se han mencionado varias ventajas de poseer un servicio de log compartido a nivel sistema operativo. Este esquema también tiene desventajas. Un servicio de log compartido no comprende el algoritmo que cada uno de los administradores de recursos utiliza y no interpreta el contenido de los registros de log. Otro problema surge del requerimiento que dice que el servicio de log debe protegerse de clientes

independientemente diseñados e implementados que pueden fallar o hacerse inaccesibles. Una consecuencia de esto es que el servicio debe estar en un espacio de direcciones (o "protection domain") separado del de los administradores de recursos que son los clientes del servicio de logging.

Esta separación crea un problema de performance para la operación de escritura bufferizada que es una de las operaciones de logging más comunes. Una escritura bufferizada agrega un registro a una memoria volátil y retorna un LSN que se usa en el futuro para recuperara el contenido del registro. Los administradores de log de las bases de datos convencionales asignan la dirección relativa del byte como LSN, haciendo muy eficientes las operaciones de lectura. Pero, si un servicio de logging común asigna la dirección relativa del byte como LSN, cada operación de escritura bufferizada agrega el overhead de comunicación interproceso o llamadas a través de dominios (interprocess communication or cross domain calls). Este overhead es considerable en varios sistemas. Los servicios de log compartido necesitan proveer registros bufferizados en los administradores de recursos, lo que complica la asignación del LSN y la implementación de la operación de lectura.

Un servicio de log común maneja un espacio finito de logging que es compartido por múltiples clientes. El servicio de log reutiliza este espacio cuando los clientes indican que los datos no son necesarios para recuperación o que se puede conseguir offline para recuperación frente a fallas de los medios de almacenamiento. Cuando los logs de diferentes clientes se encuentran entremezclados en el almacenamiento persistente (como es necesario en ocasiones para obtener mejoras en la performance) luego todos los logs de todos los clientes deben liberarse antes que el almacenamiento pueda ser reutilizado. Pero puede ocurrir que los clientes administrados en forma independiente no liberen todos los registros o puede que no estén disponibles para responder a los pedidos del servicio de log para que los libere. Los servicios de log compartidos deben tener mecanismos para reclamar espacio de log aún cuando los administradores de recursos (los clientes) no lo pueden hacer o no pueden responder. Los mecanismos de reclamo deben mover los registros de lugar y esto invalida el uso de la dirección relativa del byte como LSN.

Administradores de recursos independientes pueden tener diferentes requerimientos relativos a la confiabilidad de sus datos de log. Por ejemplo, un sistema de archivo que sólo hace logging de meta-datos y recupera de backups periódicos después de una falla de disco no necesitará doble log de datos, mientras que un DBMS almacenando datos críticos necesitará un doble log. Es difícil para un servicio de log proveer estos requerimientos variantes y a la vez proveer las ventajas de un log compartido. El uso de streams de log independientes puede complicar la administración de transacciones comunes.

En el Almaden Research Center de IBM, se está implementando un nuevo Administrador de Log para el sistema operativo distribuido QuickSilver. El Administrador de Log se refiere a los problemas de una asignación eficiente de LSN y

reclamo de espacio de log. Modificaciones al administrador de transacciones de QuickSilver se referirán al problema de variar los requerimientos de confiabilidad.

Como todos los sistemas, salvo almacenamiento de procesos a bajo nivel e IPC, el Administrador de Log de QuickSilver corre como un proceso separado y se comunica con los clientes y otros servidores del sistema por medio de IPC. El Administrador de Log no impone algoritmos determinados de recuperación a sus clientes, a pesar que su diseño da especial consideración a los esquemas de logging de los algoritmos comunes de recuperación de base de datos. Se usan enteros ordenados de manera arbitraria como LSN. Esta estrategia de asignación de LSN evita los problemas frecuentes de IPC y reclamo de espacio de log arriba mencionados.

El problema del costos de IPC entre administradores de recursos y el Administrador de Log se maneja a través de la bufferización de registros en las rutinas de librería de los administradores de recursos (los clientes). Cuando un cliente escribe un registro, un LSN se asigna a través de una rutina de librería. Los nuevos registros se bufferizan inicialmente en el espacio de direcciones del cliente. Luego los registros se flushean hacia la memoria volátil del Administrador de Log o hacia el almacenamiento persistente. Cuando un registro de un archivo de log de un administrador de recursos se flushea a memoria persistente todos los registros previamente flusheados de otros archivos de log de otros administradores de recursos en el mismo grupo de log también son forzados a almacenamiento persistente. De esta manera una única operación de "force" del Administrador de Transacciones de QuickSilver puede hacer commit todos las modificaciones que fueran logueadas por los administradores de recursos si sus archivos de log están en el mismo grupo.

El Administrador de Log de QuickSilver usa una combinación de "call back requests" y archivos especiales para solucionar el problema de administrar el espacio de log compartido. El espacio para log es reclamado cuando los administradores de recursos liberan registros de log o cuando los registros se han guardado en almacenamiento offline o online. Los administradores de recursos proveen rutinas de "call back" para que el administrador de log pueda solicitar que se libere espacio de log. El mecanismo de archivo incluye filtros especiales de los administradores de recursos, que especifican que registros archivar. Existe un archivo de "online crash recovery" que contiene registros necesarios para transacciones que tienen un largo tiempo de ejecución y para la recuperación de administradores de recursos que no responden para cuando se hagan accesibles. De esta manera el administrador de log es capaz de reclamar espacio de log conteniendo registros de administradores de recursos que no responden, moviendo esos registros al archivo de "online crash recovery".

El problema de los administradores de recursos con diferentes requerimientos de confiabilidad se trata permitiendo a diferentes administradores de recursos tener archivos de log asignados a diferentes grupos de archivos de log con diferentes niveles de confiabilidad. Los diferentes grupos de archivos de log no pueden forzarse simultáneamente, pero el administrador de transacciones de QuickSilver escribirá

registros de commit en ambos grupos de archivos de log. El administrador de transacciones realiza un two phase commit para asegurar la consistencia de la transacción.

El administrador de log de QuickSilver es complejo. Contiene una base de datos interna que utiliza un log para recuperación. El administrador de recursos le provee servicio a múltiples administradores de recursos, eliminando la necesidad de que cada administrador de recursos implemente su propio log. El servicio provee considerables funciones comparadas con el servicio de log sencillo que cada uno de los administradores de recursos podría implementar por su cuenta. El diseño del administrador de log de QuickSilver se acomoda a un amplio rango de tamaños de sistemas. Los clientes de log tienen gran flexibilidad al determinar la confiabilidad de los datos del log.

5.6.3 Diseño del mecanismo de logging para los managers.

En la sección anterior se brinda el mecanismo de logging como un servicio del sistema operativo que podría ser utilizado por los managers (desde el punto de vista de pedido-espera).

Nuestro esquema maneja dos niveles de logging: usamos el syslog del sistema operativo UNIX para logging de mensajes y funcionamiento general del sistema, y utilizamos un esquema propio como almacenamiento de las operaciones que no deben perderse ante una caída. Este esquema se basa en mecanismos de logging de las bases de datos, pero poniendo especial cuidado en el tema performance y teniendo en cuenta la restricción que todas las operaciones deben ser atómicas. Esto se diferencia de las bases de datos que manejan un número de operaciones a las que se les puede hacer rollback, si no fuera así y por cada operación debieran acceder a disco, la performance bajaría substancialmente. Este número es parametrizable y modificable por el administrador de base de datos que debe tomar una decisión basándose en la relación costo-beneficio (persistencia-performance).

Los managers proveen el mecanismo de logging a nivel de la aplicación, ya que el sistema operativo no lo provee. De esta manera se cumple el objetivo de poder utilizar la herramienta en sistemas operativos ya existentes y que no proveen este servicio. Hasta tanto todos los sistema operativo no provean este tipo de servicios se puede utilizar una aplicación como los managers para ubicarse entre el sistema operativo y las aplicaciones de usuario, brindando a estas últimas los mismos servicios que debería proveer el sistema operativo. Esto constituye un middleware que permite a las aplicaciones independizarse del sistema operativo sobre el cual ejecutan.

Es fundamental para el diseño de los managers que se asegure la persistencia de los trabajos manejados. Se utiliza este esquema de logging como almacenamiento persistente para los mismos, debiendo cumplir el requerimiento de no perder ninguna operación relevante realizada sobre los trabajos.

Las funciones provistas por el logserver son utilizadas para garantizar la robustez del sistema almacenando los datos en forma persistente. Los managers deben retornar siempre a un estado consistente. No se puede perder ninguna operación relevante. TomarToken no es una operación relevante ya que de interrumpirse sólo se perdería la ejecución de alguna aplicación, pero no un dato del sistema -un trabajo-. Todas las demás operaciones de los clientes son relevantes y deben ser logueadas, porque conllevan modificaciones en el estado de los trabajos administrados.

El esquema de logging de los managers se diferencia substancialmente de aquellos usados para las bases de datos en la manera en que agrega registros al log. La diferencia responde a una distinta necesidad: no se debe perder nunca ningún registro de log. En el modelo de log usado para las bases de datos, no siempre que se escribe un registro de log inmediatamente, este es almacenado en forma persistente, existiendo la posibilidad de pérdida de registros de log. Esta posibilidad es mínima y depende de cada cuánto se realiza la operación de force. Por los requerimientos de nuestro modelo se debe realizar una operación de force por cada operación de escritura. Las operaciones de escritura y force se han debido convertir en una única operación: cada force escribe solamente un registro de log -aquel sobre el que se realiza la operación de escritura. Es importante notar que esto tiene una incidencia muy negativa sobre la performance. Por esta razón diseñamos un esquema donde se cuida principalmente la performance en cada detalle de implementación. Para obtener una mayor velocidad de respuesta se mantiene una copia en memoria volátil y sobre ésta se realiza la lectura. Cuando se estudie el diseño y la implementación del esquema de logging se verán en detalle los cuidados tomados para alcanzar una muy buena performance y no tener posibilidad de pérdida de ningún registro de log.

A los efectos de la explicación anterior, se podría considerar que cada una de las instancias de los servidores del esquema juega el papel de un administrador de recursos. Cada una de las instancias de los servidores debe loguear sus operaciones relevantes y para esto utiliza al administrador de log que es único y compartido en el sistema, adquiriendo las dificultades de un servicio de log compartido.

Con respecto a la operación de escritura bufferizada, ya se ha mencionado que no existe como tal sino que se fuerza la escritura en un medio persistente por cada operación de escritura. El log se usa solamente para recuperación ante caídas. La escritura en el log persistente se realiza en forma ordenada. Por estas razones no asignamos LSN a los registros de log, ya que no se lee de ellos en forma random y los registros se hallan ordenados. En caso de caída todo el log se procesa en forma secuencial y en orden de escritura de los registros.

Se utiliza un formato especial de escritura elegido atendiendo a la performance. El tratamiento y proceso que se realiza al log para recuperar la información que había en las tablas de memoria volátil será explicado al presentar el diseño e implementación del servicio de log usado por los servidores.

Compactación de los archivos de log.

El mecanismo utilizado por el administrador de log de los managers para recuperación de espacio de log es diferente al tratado en la sección precedente. Se basa en tener un número determinado de buffers de log que son manejados en forma circular. Todos los buffers se encuentran en almacenamiento persistente.

Eventos específicos disparan la transformación de estos logs que son escritos en un formato especial en otro log compactado, también estacionario. Así se liberan los buffers de log circulares, liberándose el espacio de log sobre el que se continúan escribiendo registros. Se pasa de tener un histórico de movimientos de trabajos entre colas a tener un estado del sistema. Este último es resumido y elimina los estados intermedios de trabajos entre colas. El historyserver es el servidor de los managers encargado de cumplir esta tarea. Su funcionamiento y características se tratarán en detalle en la sección referida al historyserver.

El esquema descripto no presenta varios niveles de confiabilidad, porque según los requerimientos de los servidores, el administrador de log debe ser completamente confiable y no perder jamás ningún registro de log.

El modelo del administrador de log de los managers se mantuvo simple deliberadamente. Se trata de proveer un esquema de log para ser usado únicamente durante la recuperación ante caídas de los managers. Los únicos clientes de este servicio son las diferentes instancias de los procesos servidores que ejecutan un lock por cada una de las operaciones que cambian el estado de los trabajos en el sistema. Corre en un espacio de direcciones diferente a sus clientes, pero en una misma máquina y se comunica con ellos a través de IPC. Debe ser rápido, eficiente y completamente confiable.

5.6.3.1 Estructura de datos.

Este servidor tiene sus propias estructuras de datos para almacenar temporariamente los datos enviados por sus clientes, hasta almacenarlos en un medio persistente. El logserver no comparte estructuras de datos con el resto de los servidores de la herramienta.

5.6.3.2 Estructura del logserver.

El logserver no es un servidor de RPC. A diferencia de los servidores como connectserver, getserver y putserver, el logserver es un proceso que forma parte de los managers pero que no presta servicios a los clientes de la herramienta. No posee ninguna función que sea invocada por los clientes de la misma. El logserver presta servicios a los servidores que integran la herramienta como el connectserver, getserver y putserver.

El logserver es un proceso que ejecuta como un demonio de Unix. Las funciones del logserver son invocadas por los servidores antes mencionados.

Se sincroniza con el historyserver mediante el uso de señales. El logserver le avisa al historyserver mediante una señal que se llenó un buffer y que puede procesar los buffers de log.

5.6.3.3 Acciones realizadas por el logserver.

La función única del logserver es la de almacenar en forma persistente las operaciones que le son enviadas por los integrantes de los managers en forma de mensajes. El logserver es un servidor, y como tal, tiene una etapa de inicialización y un ciclo de vida infinito recibiendo los pedidos de sus clientes y realizando el servicio solicitado. Efectúa las siguientes acciones:

1. Recibe como parámetros el identificador de la cola de mensajes creada por el connectserver y que servirá para comunicar a todos los integrantes de los managers, el identificador de proceso (pid) del servidor de historia y los identificadores de los semáforos asociados a cada uno de los archivos de logging.
2. Abre los archivos de logging. Estos fueron previamente creados por el connectserver. De producirse cualquier error al abrir estos archivos, el servidor hace un *exit* provocando la muerte en cascada de los managers. De esta manera se asegura el no continuar procesando si no se pueden almacenar las operaciones realizadas. Ejecutar sin logging provocaría manejar información incompleta o incosistente.
3. Se ejecuta la función de entrada al servidor llamada *server()*. Esta es un ciclo infinito que recibe las solicitudes de servicio.
4. Cierra los archivos de logging. Esta acción no llega a ejecutarse cuando no se producen errores.

5.6.3.4 Acceso de los clientes al logserver

El servidor de logging actúa como servidor para todos los demás procesos que conforman los managers. Cada uno de los servidores de la herrarmienta es un cliente para el servicio que presta el logserver.

La función usada por los clientes del logserver para acceder a sus servicios se denomina *writelog* y es la encargada de enviar un mensaje al logserver conteniendo la operación a ser almacenada en el archivo de logging para persistencia. El cliente que llama a esta función se bloquea hasta no recibir el consentimiento del servidor. Cuando *writelog* retorna, ya se ha salvado en almacenamiento persistente la operación.

5.7 Diseño del servidor de historia: historyserver

El objetivo de este servidor es soportar y garantizar la recuperación ante fallas del sistema, trabajando en cooperación con el logserver.

Este módulo es el responsable de aplicar la semántica asociada a un movimiento de trabajos, a los archivos que deja el servidor de logging. Si un trabajo ha sido pasado de la cola X a la cola Y, hay que eliminar todas las ocurrencias que ubican al trabajo en la cola X y almacenar la información que indica que el trabajo está en la cola Y.

5.7.1 Diseño

El servidor de historia o historyserver trabaja en combinación con el logserver para brindar un sistema de recuperación ante fallas. El logserver registra en almacenamiento persistente, sean archivos o raw device, las transacciones que deben persistir ante caídas. Una vez que se alcanza la capacidad total de uno de estos almacenamientos (buffer), el logging de las transacciones sigue en otro buffer (archivo o sección del raw device). El buffer lleno se libera para ser procesado por el historyserver.

Existe información que debe ser persistente aún en caso de caídas o fallas. Esta debe ser tratada en forma segura para poder garantizar la restitución del estado previo a la caída. Dicha información es la que registra qué trabajos se encuentran dentro del flujo del sistema y en qué colas se encuentran encolados. Es la única información fundamental para restaurar el sistema en caso de pérdida de las estructuras de memoria que soportan el funcionamiento de la herramienta.

El logserver simplemente registra por cada transacción el par <Trabajo, Cola>. El servidor de historia se encarga de mantener un archivo histórico que constituye una imagen del estado del sistema en cada momento. La imagen se modifica cada vez que se procesa un nuevo buffer para incorporar la información registrada en el mismo en el archivo de historia.

El historyserver comienza su trabajo cuando el logserver le avisa que ha llenado un buffer y que el mismo está disponible para ser procesado. El procesamiento consiste en leer cada par <trabajo,cola> contenido en el archivo de historia y aplicar sobre el nombre de cada trabajo una función que calcule una clave de hashing. Esta clave es la usada para acceder a la tabla de hashing que almacena los pares <trabajo, cola>. Como sólo es necesario almacenar la última posición de un trabajo en una cola, a medida que se procesa el archivo de historia, se modifica la dupla de la tabla de hashing para contener la última cola que contiene al trabajo. Al finalizar el procesamiento del servidor, los pares mencionados en la tabla de hashing representan la imagen del sistema.

En la información contenida en la tabla de hashing se consolidan dos archivos: el primero es la imagen de la historia al momento de producirse el procesamiento del historyserver, y el segundo es el buffer a procesar. El orden cronológico de los movimientos va del archivo de historia (lo más viejo) a el buffer recién completado (lo más nuevo). Primero se levantan a la tabla de hashing los datos del archivo de historia. Una vez que todas las entradas del mismo se encuentran en la tabla de hashing, se procesa el buffer del logging. A ambos se les aplica la misma función para el cálculo de la clave de hashing y para el mismo trabajo la función resuelve a la misma posición de la tabla. Si el logserver registró el trabajo "t" en la cola "X", en la tabla de hashing se encuentra la dupla <trabajo "t", Cola "X">. Si al leer el buffer se encuentra la dupla <trabajo "t" Cola "Y">, se calcula la función de hashing de "t" y se cambia el valor de la cola de la dupla <trabajo "t", Cola X> de la tabla de hashing por la dupla <trabajo "t", Cola "Y">. De esta forma esa entrada de la tabla de hashing refleja el último estado leído de dicho trabajo. Al finalizar de procesar los buffers, se obtiene el último estado de dicho trabajo.

Para mantener la consistencia de la información es fundamental leer el buffer en el orden de producidos los movimientos de trabajos. De esta forma se aplica la semántica de los movimientos de los trabajos en las diferentes colas a través del tiempo. Por cada par <trabajo, cola> que se lea del buffer, se busca en la tabla de hashing si existe una ocurrencia previa de dicho trabajo. De existir, significa que este trabajo ha cambiado de cola y hay que actualizar la información de la dupla de la tabla de hashing con los datos de la nueva cola. Si el trabajo no se encuentra en la tabla de hashing, significa que se ha creado un nuevo trabajo y que fue introducido al sistema en la cola especificada debiendo insertar una nueva dupla en la tabla de hashing.

Este mecanismo de búsqueda en la tabla de hashing a partir del cálculo de la clave sobre el nombre del trabajo, garantiza que no haya trabajos repetidos.

Dentro de un mismo buffer es posible registrar varios movimientos de un mismo trabajo, debiéndose almacenar en la tabla de hashing el dato que se encuentra asociado a la última ocurrencia del trabajo en el buffer representado por la última cola donde fue depositado.

Una vez procesadas todas las entradas del buffer, la tabla de hashing refleja la nueva imagen del sistema y debe ser salvada en un archivo persistente para ser utilizada en caso de caídas.

Para tratar los trabajos que son *borrados* del sistema, se tienen al menos dos alternativas de diseño. La primera consiste en no ingresar la dupla que contiene como cola el identificador de borrado a la tabla de hashing y si existiera una ocurrencia previa del trabajo hay que borrar la dupla de la tabla de hashing. La segunda alternativa consiste en tratar dicho trabajo como cualquier otro en el momento de procesar el buffer utilizando para ello un identificador de cola especial indicando que el trabajo ha sido borrado. Cuando se termina de procesar el buffer y se procede a salvar el nuevo estado en un archivo persistente, todos aquellos trabajos que tengan como cola el identificador de borrado, se saltean y no son grabados, eliminándolos del estado actual.

Se optó por esta segunda alternativa ya que permite tratar todas las duplas de la misma manera al momento de armar la tabla de hashing, homogeneizando los procesos realizados sobre las mismas y evitando el proceso costoso de eliminación en las listas de la tabla de hashing.

Es necesario que una vez que el nuevo estado se encuentra correctamente salvado en un archivo persistente se pueda, en caso de falla, recuperar el estado de ejecución de la herramienta a partir del mismo. Por este motivo, el connectserver, cada vez que comienza su ejecución, invoca un procedimiento que restaura el estado de las tablas del sistema a partir del archivo de historia. Utilizándose una tabla de hashing, se leen todas las uplas <trabajo, cola> existentes en el archivo persistente de historia, y de existir, se procesan los buffers de logging que se estaban procesando pero estaban incompletos al momento de la caída. De esta forma se restaura el estado del sistema y se procede a inicializar la Tabla de Trabajos con estos datos.

5.7.1.1 Estructura de datos.

Este servidor tiene sus propias estructuras de datos para almacenar temporariamente los datos leidos de los archivos de historia y hasta almacenarlos en un medio persistente nuevamente. El historyserver no comparte estructuras de datos con el resto de los servidores de la herramienta.

5.7.1.2 Estructura del historyserver.

El historyserver no es un servidor de RPC. El historyserver es un proceso que forma parte de los managers pero que no presta servicios a los clientes de la herramienta. No posee ninguna función que sea invocada por los clientes de la misma.

El historyserver ejecuta como un demonio de Unix y corre independientemente de los servidores que son servidores de RPC. Se sincroniza únicamente con el logserver mediante el uso de señales.

Este servidor ejecuta cuando el logserver le avisa mediante una señal que se llenó un buffer y que, por lo tanto, el historyserver tiene trabajo que realizar. El historyserver no exporta funciones para que sean invocadas por otros procesos, sino que realiza sus tareas dentro del manejador de señales.

5.7.1.3 Acciones realizadas por el historyserver.

Este módulo consta de cuatro funciones principales.

Main()

El Main() es la función encargada de manejar la sincronización con el servidor de logging, y de iterar en espera de buffers a procesar. Para establecer la sincronización entre el servidor de historia y el de logging, se utiliza el mecanismo de señales que provee Unix. El historyserver se bloquea en espera de una señal generada por el logserver para indicarle que se completó un nuevo buffer y que debe ser procesado. Se diseñó un esquema con dos buffers, que puede ser fácilmente generalizado a un esquema con número variable de buffers. A cada buffer se le asocia un semáforo para asegurar la exclusión mutua en el acceso entre el servidor de logging y de historia. Podría suceder que el logserver, haya completado el segundo buffer sin que el historyserver haya terminado de procesar el primero. En este caso se debe esperar a que se procese el primero antes de continuar registrando transacciones. Para bloquear a los servidores se utiliza el semáforo asociado al buffer. Todos se bloquean esperando que el logserver pueda registrar sus transacciones. Esto no debería suceder nunca o en casos muy excepcionales, ya que generaría un alto en el proceso de administración de los managers, incapaces de responder a los requerimientos de los clientes. De ocurrir frecuentemente se debe incrementar la cantidad de buffers de Logging.

CompressHistory

La función denominada CompressHistory lee las duplas <trabajo, cola> del archivo de historia y de los buffers y arma consistentemente la tabla de hashing, reflejando los movimientos que se fueron produciendo en el sistema. La función recibe como parámetros la tabla de hashing y el archivo de historia o buffer que le corresponde procesar. Existe más de un buffer y previo a cada buffer hay que procesar el archivo de historia existente hasta ese momento.

FlushHistory

Como complemento a CompressHistory, una vez que se termina de procesar el buffer y se completa la tabla de hashing la información se almacena en un archivo persistente para mantenerse disponible en caso de fallas. La función que realiza esta tarea es FlushHistory y recibe como parámetros la tabla de hashing y el archivo persistente donde se deberá salvar la información.

PonerEnBlanco

Una vez que se el nuevo estado fue satisfactoriamente salvado y no hay peligro de perder información, se procede a blanquear el buffer de logging recién procesado de manera tal que el servidor de logging pueda comenzar a registrar las transacciones en el mismo. La función que se encarga de inicializar nuevamente el buffer es PonerEnBlanco y obtiene como parámetro el identificador del buffers a blanquear. Puede utilizarse sobre archivos o secciones de un dispositivo crudo (raw device).

5.8 Diseño del servidor de administración: admserver

El admserver es un servidor de RPC.

Provee la función Present utilizada por un nuevo cliente de la herramienta para registrarse y acceder a los servicios de la misma.

El *admserver* es el encargado de manejar funciones de administración de los managers. Provee los servicios necesarios para que un operador pueda conocer el estado de los managers, las colas y procesos que estos manejan.

También se proveen las funciones de dar de baja un cliente o liberar los recursos de todos los clientes de una máquina que falló.

5.8.1 Diseño

Este servidor provee dos clases de servicios: el primero está orientado a los clientes de la herramienta y el segundo tipo de servicio está destinado a los administradores y operadores del sistema administrado por la herramienta.

El admserver es el servidor de los managers al que los clientes se conectan primero. Este servidor, mediante la función Present, permite a los clientes registrarse para el uso futuro de los managers.

Este servidor provee todas las funciones necesarias para monitorear y administrar la herramienta y los objetos que la misma maneja. Como las estructuras donde se almacenan el estado de los procesos, las colas y los trabajos son tablas en memoria con un sistema restringido de acceso, se diseñó un servidor interno a la herramienta para proveer las funciones de monitoreo y administración. Así se asegura la no interferencia de estas tareas con las de trabajo normal de la herramienta. Se soportan dos tipos de funciones de administración: aquellas que causan cambios en el estado de los managers y aquellas que no. Como ejemplo de las funciones que pueden variar el estado se encuentra dar de baja un cliente o dar de baja una máquina donde ejecutan clientes de los managers. Como ejemplos de la segunda se puede mencionar conocer todos los trabajos encolados en determinada cola, conocer los procesos clientes ejecutando en determinada máquina, y otras funciones de monitoreo.

Además de este servidor que realiza funciones de administración, la herramienta provee un proceso que es un cliente de los managers que permite realizar operaciones como agregar, borrar y cambiar de ubicación trabajos en las colas. Este cliente con funciones administrativas se denomina *admclnt*. Referirse a la descripción del mismo para más detalles.

5.8.1.1 Estructura de datos.

Las estructuras de datos utilizadas por el admserver son las estructuras detalladas en la sección 5.2.1.1 "Diseño de las estructuras de datos", página 46 y otras estructuras privadas del mismo.

El admserver utiliza las estructuras de datos que almacenan información relativa a clientes, trabajos, colas y tipos de procesos.

5.8.1.2 Estructura del admserver.

El admserver es un servidor de RPC.

El punto de entrada para el mismo es la función ParseArgsAdm. Esta función se ejecuta cuando comienza este servidor y se encarga de parsear los argumentos que le pasa el connectserver al admserver en el momento de realizar el fork del mismo. El admserver obtiene los identificadores de las tablas en memoria compartida que comparte con los demás servidores que conforman los managers. Esta función inicia el logging en el syslog.

Una vez ejecutada la función ParseArgsAdm, el admserver se transforma en un servidor que provee los servicios que se describirán más abajo.

5.8.1.3 Acciones realizadas por el admserver.

Este módulo consta de ocho funciones de monitoreo y administración, una que permite que nuevos clientes se registren y una de inicialización llamada ParserArgsAdm.

Conexión de un cliente a la herramienta.

La función Present recibe el pedido de un cliente nuevo que desea hacer uso de la herramienta y lo da de alta en el sistema. Para incorporarlo al sistema se requiere conocer el host o máquina donde ejecutará el cliente y el tipo de proceso que este cliente utilizará. El tipo de proceso determina a qué colas tendrá acceso el cliente.

Borrado de un cliente.

La función ClientDown es invocada cuando un cliente de la herramienta muere. El mecanismo implementado por monitor/watchdog permite conocer esta situación. La función ClientDown es la encargada de actualizar las tablas de los managers para liberar todos los recursos del cliente. Es importante cambiar el identificador de la entrada en la tabla lo antes posible para evitar inconsistencias.

Información ordenada por tipo de proceso.

La función DumpTablesCl pertenece a la categoría de servicio de administración. Mediante esta función es posible acceder a las tablas de memoria y obtener la

información de todos los clientes conectados con un determinado tipo de proceso, los trabajos que tienen tomados y la cola a la que pertenece cada trabajo.

Información de colas.

La función DumpTablesQ pertenece a la categoría de servicio de administración. Mediante esta función es posible acceder a las tablas de memoria y conocer las colas existentes en el esquema.

Se recorre la tabla de colas y para cada una de ellas se registra el nombre y el identificador.

Información de tipos de proceso.

La función DumpTablesProc pertenece a la categoría de servicio de administración. Mediante esta función es posible acceder a las tablas de memoria y conocer los tipos de proceso existentes en el esquema.

Se recorre la tabla de procesos y para cada una de ellos se registra el nombre.

Existencia de un trabajo determinado en una cola.

La función ExisteToken pertenece a la categoría de servicio de administración. Mediante esta función es posible conocer si un determinado trabajo existe en una cola especificada.

Búsqueda de un trabajo.

La función BuscarTokenEnQueues pertenece a la categoría de servicio de administración. Mediante esta función es posible conocer si un determinado trabajo existe en el esquema y averiguar en qué cola se encuentra. A diferencia de la función ExisteToken, busca en todas las colas definidas en el esquema.

Búsqueda de un conjunto de trabajos.

La función VerLote pertenece a la categoría de servicio de administración. Mediante esta función es posible obtener una lista de todos los trabajos de una cola cuyos nombres cumplen una determinada condición. Se le pasa como parámetros el pattern que debe cumplir el nombre del trabajo y el nombre de la cola.

Cantidad de trabajos por cola.

La función ContarJobs pertenece a la categoría de servicio de administración. Mediante esta función es posible conocer la cantidad de trabajos encolados en una determinada cola.

5.9 Diseño del módulo monitor.

El monitor, en colaboración con el watchdog, se encarga de monitorear el funcionamiento de los clientes de la herramienta. En caso de muerte de algún cliente libera los recursos tomados por el mismo y devuelve los trabajos en proceso a las colas de donde fueron tomados. Devolver los trabajos posibilita que otro cliente, con acceso a esa cola, pueda procesar el trabajo incompleto.

5.9.1 Diseño.

El monitor trabaja en combinación con los watchdogs, que ejecutan en cada máquina cliente, para actualizar el estado de los clientes ante falla o muerte de los mismos.

Es fundamental que los managers puedan reconocer cuando un cliente ha dejado de funcionar para retornar los trabajos tomados al flujo normal de procesamiento.

El monitor verifica, a cierto intervalo de tiempo, que todos los clientes que tiene registrados siguen operando. Para ello colabora con los watchdogs locales a las máquinas donde ejecutan los clientes a monitorear. Debido a que pueden producirse fallas en la comunicación, en las máquinas clientes o hasta en los procesos watchdog mismos es difícil mantener un esquema confiable para registrar fallas de los clientes. Se debe proveer un mecanismo capaz de reconocer fallas por omisión y de performance. Ver sección 4.3.2 "Tolerancia a fallas" para mayor detalle referido a clasificación de fallas.

Es necesario reconocer instancias de clientes anteriores a la era de los actuales managers o watchdogs. Podría darse el caso que los managers son terminados pero no todos los clientes conectados a los mismos mueren. Cuando se levanta la nueva instancia de los managers se debe reconocer a los clientes anteriores y negarles el uso de la herramienta. Estos clientes deben reconectarse a los managers para conseguir una conexión válida, renovando sus permisos de acceso.

El monitor participa en el mecanismo de presentar un cliente a los managers. Para mayor detalle del mismo referirse a la sección 5.10 "Esquema de la conexión de un cliente de la herramienta con los managers". El monitor solicita al admserver que se otorgue el alta a un nuevo cliente, recibe el identificador que entrega el admserver y se lo comunica al watchdog ejecutando en la máquina del cliente. El watchdog entrega el identificador al nuevo cliente.

El monitor se comunica con los watchdogs usando el protocolo TCP/IP. [Tanenbaum 008] [Comer 009].

5.9.1.1 Estructura de datos.

El monitor utiliza sus propias tablas donde almacena los datos referidos a los clientes conectados con los managers y las máquinas donde ejecutan estos clientes. El monitor maneja dos tablas: de hosts (o máquinas) y de clientes.

La tabla de máquinas contiene la lista de las máquinas donde ejecutan clientes de la herramienta. Para cada una almacena el identificador de proceso del watchdog ejecutando en dicho host, el nombre del host y un puntero a la lista de clientes ejecutando en el mismo.

Para cada cliente almacena el identificador del mismo dentro de la herramienta.

Los procesos que conforman el monitor comparten las tablas de datos que se ubican en memoria compartida para tal fin. Se usan semáforos para sincronizar el acceso a las mismas.

5.9.1.2 Estructura del monitor.

A diferencia de servidores como connectserver, getserver y putserver, el monitor no es un servidor de RPC y no presta servicios en forma directa a los clientes de la herramienta. No posee ninguna función que sea invocada directamente por los clientes de la misma.

El módulo monitor está integrado por tres procesos. El proceso *monitor* es el padre de los otros dos procesos llamados *assign* y *ping*. Los tres procesos ejecutan en cooperación.

Los procesos que integran el monitor ejecutan como demonios de Unix y corren independientemente de los servidores de RPC de la herramienta. Se comunica con los watchdogs a través del protocolo TCP/IP e invoca funciones del admserver vía RPC. [Tanebaum 008] [Comer 009].

5.9.1.3 Acciones realizadas por el monitor.

Como se explicó, el módulo monitor está integrado por tres procesos. Organizaremos esta sección describiendo cada uno de los mismos y su interacción.

Todos los procesos que integran el monitor loguean sus acciones y estados permitiendo el debbugging del accionar de los mismos. Esto es importante por el esquema de procesos clientes distribuidos en distintas máquinas que se desea monitorear. Se soportan diferentes niveles de debugging parametrizables que producen distinto nivel de detalle en la información loggeada.

5.9.1.3.1 Monitor

El proceso monitor se encarga de crear los otros dos procesos que conforman el módulo monitor y liberar los recursos de los mismos al terminar la ejecución, sea en

forma normal o anormal. Los procesos assign y ping son los que realizan el trabajo específico de monitorear a los clientes.

La siguiente es una lista de las funciones llevadas a cabo por el monitor. No constituye una secuencia de acciones o ejecución.

1. Abre el archivo de log donde de almacenan las acciones y estados relevantes del monitor. El nivel de detalle de esta información es parametrizable.
2. Establece el manejador de señales compartido por los tres procesos para la señal SIGHUP. El manejador de señales resetea el monitor. Resetear el monitor significa borrar toda la información de clientes y máquinas contenida en las tablas del monitor.
3. El monitor se presenta a los managers para convertirse en un cliente de RPC del admserver.
4. Crea las tablas de memoria compartida utilizadas por los tres procesos y las inicializa.
5. Crea el semáforo para sincronizar el acceso de los tres procesos a las tablas.
6. Realiza el fork del proceso que ejecuta la función assign.
7. Realiza el fork del proceso que ejecuta la función ping.
8. Inicializa el manejador de señales para las señales SIGTERM o SIGINT. De recibirlas interrumpe y termina la ejecución de los tres procesos que conforman el módulo monitor.
9. El monitor contiene el código de dos funciones que son invocadas por los procesos assign y ping. Una de ellas libera la información de un determinado cliente y la otra libera la información de todos los clientes ejecutando en una máquina determinada. Estas funciones se ejecutan al reconocerse la muerte de un cliente.

5.9.1.3.2 Assign.

El proceso assign es el proceso del monitor que participa en el mecanismo de presentación de un nuevo cliente a los managers. El assign escucha en el port en el que el watchdog escribe para solictar un Present de un nuevo cliente. Con la información que envía el watchdog local al cliente, el assign gestiona a través del admserver el alta del cliente.

Este proceso posee una función principal encargada de realizar esta tarea y otras funciones que colaboran para realizarla.

Las acciones realizadas son:

1. Realizar el attach de las tablas de memoria compartida.
2. Crear el socket para recibir.
3. En un ciclo mientras vivan los managers, recibe pedidos del watchdog para presentar un nuevo cliente y los procesa.

5.9.1.3.3 Ping.

El proceso ping es el proceso del monitor encargado de monitorear el funcionamiento de los clientes de los managers. El ping envía un mensaje en un intervalo de tiempo parametrizable a los watchdogs de las máquinas donde ejecutan clientes de los managers. Si encuentra un error en los watchdogs o los clientes se encarga de eliminar las entradas de las tablas de los mismos.

Este proceso posee una función principal encargada de realizar esta tarea y otras funciones que colaboran para realizarla.

Las acciones realizadas son:

1. Realiza el attach de las tablas en memoria compartida.
2. En un ciclo infinito mientras vivan los managers duerme durante el intervalo especificado y luego envía un mensaje a cada uno de las máquinas contenidas en las tablas del monitor y que ejecutan clientes de los managers.

5.10 Diseño del módulo watchdog.

El watchdog, en colaboración con el monitor, se encarga de monitorear el funcionamiento de los clientes de la herramienta. En caso de muerte de alguno de ellos es responsable de avisar al monitor y de esta manera posibilitar la devolución de los trabajos que el cliente estaba procesando a las colas de donde fueron tomados.

5.10.1 Diseño.

El watchdog es un proceso local a cada una de las máquinas donde ejecutan los clientes. Es el módulo de los managers que ejecuta distribuido con respecto a los servidores que integran la herramienta. Hay tantos watchdogs como máquinas donde se ejecuten clientes de la herramienta y un único watchdog por máquina.

Los watchdogs ejecutan en forma distribuida para poder monitorear, en combinación con el monitor, los clientes y actualizar el estado de los mismos ante falla o muerte. Es fundamental que los managers puedan reconocer cuando un cliente ha dejado de funcionar para así poder retornar los trabajos por él tomados al flujo normal de procesamiento.

El monitor verifica a cierto intervalo de tiempo que todos los clientes que tiene registrados siguen operando. Para ello colabora con los watchdogs locales a las máquinas de los clientes. Para conocer el estado de los clientes el monitor envía un mensaje al watchdog quien verifica el estado de los clientes y avisa al monitor en caso que alguno haya caído. Si un cliente murió, el monitor desencadena la eliminación del mismo de las tablas de los managers.

Si el watchdog no respondiera al monitor, la mera ausencia de respuesta indica al monitor que la red de comunicación falló, la máquina remota donde ejecuta el watchdog falló o el watchdog ha caído. Esta información permite al monitor desencadenar la elimininación de las tablas de los managers de todos los clientes que ejecutaban en esa máquina.

Se mencionó durante la sección 5.8 "Diseño del módulo Monitor" que es necesario reconocer instancias de clientes anteriores a la era de los actuales managers o watchdogs. Podría darse el caso que los managers son terminados pero no todos los clientes conectados a los mismos mueren. Si se levanta una nueva instancia de los managers se debe reconocer a los clientes anteriores y negarles el uso de la herramienta. Estos clientes deben reconectarse a los managers para conseguir una conexión válida, renovando sus permisos de acceso.

El watchdog participa en el mecanismo de presentar un cliente a los managers. Para mayor detalle del mismo referirse a la sección 5.10 "Esquema de la conexión de un cliente de la herramienta con los managers". El watchdog recibe el pedido de Present() de un nuevo cliente, transmite el pedido al monitor y recibe el identificador que envía el monitor otorgado por el admserver y se lo comunica al nuevo cliente.

El watchdog es el único elemento de los managers que ejecuta distribuido en la máquina de los clientes. Ejecuta en forma independiente a los demás servidores, respondiendo a los pedidos de reporte del monitor y recibiendo las llamadas de Present originadas en el cliente. Es el punto de entrada de un nuevo cliente a los managers para ejecutar el Present.

5.10.1.1 Estructura de datos.

El watchdog utiliza su propia tabla. En ella almacena los datos referidos a los clientes locales a la máquina donde ejecuta el watchdog y que están conectados con los managers.

El watchdog maneja una única tabla de clientes que almacena la lista de clientes. Para cada uno guarda el identificador del mismo dentro de la herramienta y el identificador de proceso (pid) del mismo.

La tabla se ubica en memoria propia del watchdog ya que es el único que la accede.

5.10.1.2 Estructura del watchdog.

El watchdog no presta servicios en forma directa a los clientes de la herramienta y no posee ninguna función que sea invocada por los mismos.

El watchdog es un único proceso que ejecuta en diferentes entornos y sistemas operativos. El mismo código fuente de watchdog puede recompilarse para ser ejecutado en diferentes sistemas operativos. En el presente trabajo de tesis se soportan watchdogs para ser ejecutados en varias versiones de Unix como ser AIX y Linux, y OS/2 y DOS.

El watchdog ejecuta distribuido de los managers e independientemente de los servidores de RPC de la herramienta. Ejecuta como un proceso demonio. Se comunica con el monitor y los clientes a través del protocolo TCP/IP.

5.10.1.3 Acciones realizadas por el watchdog.

La siguiente es una lista de las funciones llevadas a cabo por el watchdog. No constituye una secuencia de acciones o ejecución.

1. Si el entorno es Unix se setea el usuario efectivo de ejecución como root.
   Se inicializa el proceso como un demonio que ejecuta una única instancia en la máquina. Esta acción es realizada por una función de biblioteca que se halla en libdaemon.a. Realiza las acciones necesarias para convertir al proceso que la ejecuta en un demonio. La función es **InicializarProceso(ArchivoDeLock).**
2. Establece el manejador de las señales de SIGTERM y SIGINT para interrumpir la ejecución del watchdog.

3. Si el entorno es Unix establece el manejador de la señal SIGHUP para resetear el watchdog en caso de ser recibida. Resetear el watchdog significa eliminar todos los datos de clientes contenidos en la tabla del watchdog.
4. Averigua el nombre del host local donde está ejecutando.
5. Crea la conexión TCP y escucha sobre la misma.
6. En un ciclo infinito recibe mensajes en el socket que creó y realiza las acciones determinadas por el tipo de mensaje recibido. Las acciones solicitadas al watchdog son registrar un nuevo cliente, verificar los clientes que ejecutan en la máquina, informar el nombre real de la máquina o informar el nombre del servidor donde ejecutan los managers.

5.11 Esquema de la conexión de un cliente de la herramienta con los managers

Una vez explicados todos los módulos que integran los managers, se resume en esta sección el mecanismo de conexión de un cliente con los managers identificando todos los módulos que intervienen y nombrando las funciones que los mismos realizan. La operación de presentarse a los managers constituye una de las más complejas desde el punto de vista de la cantidad de módulos que intervienen y la forma de comunicación de los mismos. Este complejidad se debe a la necesidad de mantener la información de los clientes sincronizada y actualizada frente a fallas entre módulos que tienen diferentes funciones principales y ejecutan de manera distribuida.

En la figura se muestran los módulos que intervienen para ejecutar la función Present().

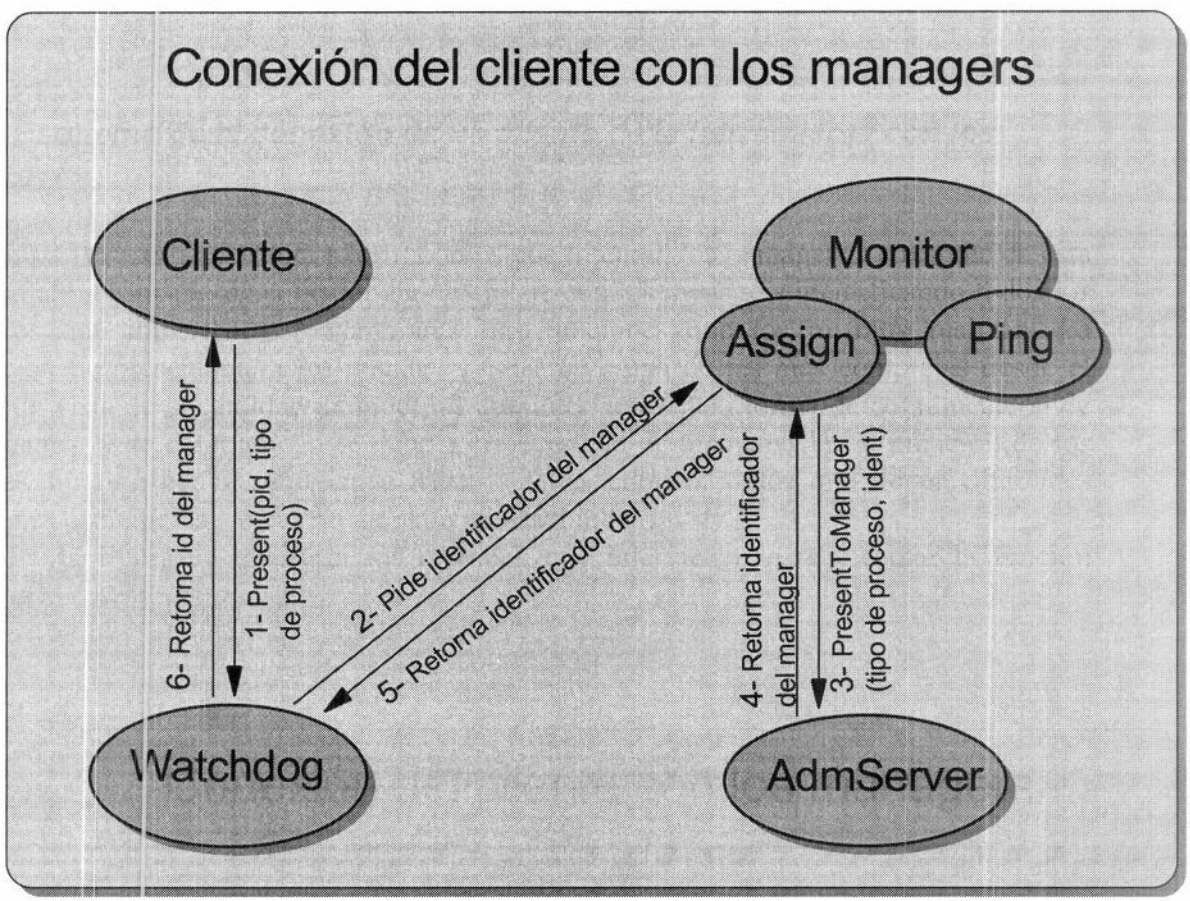


Figura 7 : Esquema de conexión de un cliente con los managers

1. El cliente ejecuta un Present() pasando la información del identificador del proceso cliente (pid) y el tipo de proceso con el cual el cliente desea conectarse a los managers. El watchdog recibe esta llamada.
2. El watchdog verifica los datos enviados por el cliente y busca espacio en sus tablas internas. Si lo obtiene envía un mensaje via TCP/IP al monitor solicitando un identificador de cliente de la herramienta. Como vimos en la explicación del monitor, el mismo está integrado por tres procesos. El assign es el proceso encargado de resolver el pedido de Present() que el watchdog efectúa al monitor. El watchdog envía un mensaje TCP a un socket que es leído por el proceso assign. El watchdog se queda esperando hasta recibir la respuesta con el identificador para el cliente.
3. El assign del monitor verifica que el watchdog sea válido y sea el mismo que el monitor tiene registrado en sus tablas internas. Si no coincidiera significa que el watchdog que ejecutaba en esa máquina murió y una nueva instancia del watchdog está en ejecución. La máquina pudo haber sufrido una falla o caída y la información almacenada en las tablas internas puede estar desactualizada. Aunque los clientes no hubieran muerto deben presentarse al nuevo manager. Por lo tanto, es necesario borrar la información de todos los clientes ejecutando en esa máquina y actualizar la información de la tabla de hosts con el identificador de proceso del nuevo watchdog. Se debe limpiar la tabla de clientes que mantiene el monitor pero también la información de los clientes que mantienen los managers en sus propias tablas. Para ello se invoca a la función ClientDown del admserver. Referirse a la sección 5.7 "Diseño del Admserver" para detalles del funcionamiento de dicha función. Al eliminarse la información de las tablas de los managers los identificadores de cliente se tornan inválidos. Si hubiera algún cliente que sobrevivió no podrá seguir operando con los managers y deberá presentarse nuevamente. Esta medida es precautoria para evitar que clientes viejos de otra corrida de los managers pudieran causar inconsistencias. Si un cliente viejo tenía trabajos en su poder y se produjo una caída o terminación normal de los managers, esos trabajos son devueltos a las colas de donde los tomó el cliente. Si luego de reiniciarse los managers ese cliente viejo deseara realizar alguna operación con ese trabajo recibirá un error porque su identificador ya no es válido.

   Si el watchdog resultó válido, el assign invoca un PresentToManager al admserver pasando como parámetro el tipo de proceso. Esta función es ejecutada por el admserver al ser invocada vía RPC.
4. El admserver otorga un identificador de cliente válido para la herramienta y registra al nuevo cliente. Le retorna el valor del identificador al assign.
5. El assign actualiza su tabla de clientes y procesos y le responde al watchdog vía TCP.
6. El watchdog actualiza su tabla interna con el identificador enviado por el assign y el pid enviado por el cliente y lo retorna al cliente de la herramienta que ejecutó el Present().

6. Implementación de la herramienta.

En el presente capítulo se detallan las decisiones de implementación consideradas para cada uno de los módulos de la herramienta, cuyo diseño fue estudiado.

El objetivo es explicar como se implementan los módulos diseñados. No se llega al más bajo nivel de detalle en este aspecto, porque no lo consideramos útil en relación al objetivo del presente trabajo de tesis de licenciatura.

6.1 Implementación del configurador

En la presente sección se describirán los detalles de implementación del configurador cuyo diseño se detalló en la sección 3 "Herramienta para definir arquitecturas" en página 24 .

El configurador es un único proceso integrado por el analizador léxico, el compilador y la rutina que los invoca y genera el archivo de configuración.

6.1.1 Primera versión: un analizador léxico y un compilador.

Como se explicó en la sección 3.1 "Diseño del configurador.", página 24, para esta versión del configurador se utilizaron las herramientas lex y yacc que provee Unix. Se escribió un proceso que utiliza el *lex* y *yacc* para generar el archivo de configuración entrada para los managers.

6.1.1.1 Analizador léxico.

El archivo *leng.l* es tomado como entrada por la herramienta *flex* (o *lex*) para generar el analizador léxico. En *leng.l* se especifican cuáles son los tokens aceptados por el lenguaje. Los tokens representan un proceso o cola en el esquema. Los nombres de los procesos comienzan con el prefijo "PROC" o "proc" seguidos de una cadena alfanumérica que represente el nombre del proceso. Los nombres de las colas comienzan con el prefijo "COLA" o "cola" seguidos de una cadena alfanumérica que represente el nombre de la cola.

6.1.1.2 Compilador.

El archivo *leng.y* es tomado como entrada por la herramienta *yacc* para generar el compilador. En *leng.y* se usa la gramática que representa el lenguaje para especificar las acciones a realizar al recibir los tokens reconocidos por el analizador léxico. Se utiliza la

gramática introducida en "Definición de un lenguaje formal". El archivo leng.y contiene la implementación de las siguientes funciones:

void parsear_grafo (char * stParam, ProcTable ppTable, ColaInterna * pcolainfo, char * prechazo, int * cc, int * cp)

Esta función es invocada desde el main ubicado en la rutina que invoca al compilador y genera el archivo de configuración. Parsear_grafo invoca al compilador propiamente dicho mediante la rutina yyparse, maneja los errores surgidos en el lexer y compilador mediante la rutina yyerror, y especifica la opción de parsear a través de múltiples archivos o no, mediante la función yywrap.

int fproc (char * res, ProcIdx * iproc)

Esta función se encarga de agregar un nuevo proceso en la tabla de procesos. Descarta aquellos que se usan varias veces en la gramática e inicializa la tabla de procesos para los procesos nuevos, con datos como ser el nombre del proceso. Mantiene un contador con la cantidad de procesos existentes en el sistema. Este valor se escribirá en el archivo de configuración.

int fcola (char * cola, int * icola)

Esta función se encarga de agregar una nueva cola a la tabla de colas. Descarta las colas que se usan varias veces en la gramática e inicializa la tabla de colas para las colas nuevas con datos como ser el nombre de la cola, el nombre del directorio donde se encuentran los datos de esa cola, así como otros valores necesarios para la ejecución de servidores RPC ya que los mismos están asociados a las colas. Mantiene un contador con la cantidad de colas existentes en el sistema. Este valor se escribirá en el archivo de configuración.

void colain (ProcIdx iproc, Qtype icola)

Esta función identifica las colas de entrada para los tipos de proceso de la gramática. Como se explicó en la descripción del lenguaje utilizado, aquellas colas ubicadas antes de un proceso en una cadena del lenguaje son colas de entrada para ese tipo de proceso. Mediante este función se inicializa la tabla de procesos con la información relativa a las colas de entrada para cada tipo de proceso definido.

void colaout (ProcIdx iproc, Qtype icola)

Esta función identifica las colas de salida para los tipos de proceso de la gramática. Como se explicó en la descripción del lenguaje utilizado, aquellas colas ubicadas después de un proceso en una cadena del lenguaje son colas de salida para ese tipo de proceso.

Mediante este función se inicializa la tabla de procesos con la información relativa a las colas de salida para cada tipo de proceso definido.

6.1.1.3 Rutina que invoca y genera el archivo de configuración.

El archivo escribe.c contiene la función main de esta versión del configurador. Es la encargada de:

1. Leer el archivo que contiene la expresión del lenguaje que representa el esquema a configurar y pasárselo al compilador.
2. Inicializar las tablas donde se almacenarán los datos.
3. Invocar al compilador.
4. Escribir el archivo de configuración.

Lleva a cabo estas acciones invocando a las siguientes funciones:

int inic_tablas( )

Esta función inicializa las tablas de procesos y colas con el formato necesario para poder ser utilizadas.

int salvar_config( )

Esta función escribe un archivo de configuración que será usado por los managers para inicializar el esquema de procesos y colas. El formato del mismo es fijo para ser fácil de reconocer por el connectserver que inicializa las tablas del sistema y determina la cantidad de servidores a ejecutar con la información almacenada en el archivo de configuración.

6.1.1.4 Ejemplo de archivo de configuración.

```
cantcolas= 4, cantproctype= 6
colarechazo=
BeginCola
C0,Harcodeado,805306372,805306373
C1,Harcodeado,805306374,805306375
C3,Harcodeado,805306376,805306377
C5,Harcodeado,805306378,805306379
EndCola
BeginProceso
P0
 s: C0
fin
P1
 e: C0, s: C1, s: C3
fin
```

```
P2
 e: C1
fin
P4
 e: C3, s: C5
fin
P5
 e: C5
fin
P3
 s: C3
fin
EndProceso
```

6.1.2 Segunda versión : interactiva.

Como se explicó en la sección 3.1 "Diseño del configurador.", página 24, esta versión del configurador está orientada al usuario final. Permite realizar altas, bajas y modificaciones de los objetos que integran el esquema de una manera más amigable, mostrando mensajes explicativos en caso de errores, pidiendo confirmación ante modificaciones o borrado de objetos, y otras características de aplicaciones orientados a usuarios finales. Para poder ser ejecutada desde una terminal no gráfica se implementó usando cursores de Unix. Es un proceso Unix que interactúa con el operador a través de una terminal de caracteres. Se eligió una terminal no gráfica para permitir la ejecución del configurador desde cualquier tipo de terminal.

La funcionalidad de esta versión incluye la anterior y agrega la posibilidad de realizar modificaciones de diversa índole a un esquema ya existente mediante el uso de menúes de ABM de los objetos y las propiedades de los mismos.

El configurador se encarga de:

1. Inicializar las tablas.
2. Leer el archivo de configuración existente.
3. Inicializar las variables utilizadas por los cursores para interactuar con la terminal.
4. Presentar el menú principal.
5. Escribir el nuevo archivo de configuración.

Lleva a cabo estas acciones invocando a varias funciones. No se presentarán las funciones encargadas de interactuar con la terminal ni presentar menúes por no estar relacionadas con la funcionalidad de la herramienta, sólo se utilizan para facilitar la interacción con la misma.

6.1.2.1 Funciones principales

1) int inic_tablas( )

Esta función inicializa las tablas de procesos y colas con el formato necesario para poder ser utilizadas. Es la misma función utilizada en la versión anterior del configurador.

2) int leoconfig( )

Esta función es la encargada de inicializar las tablas de procesos y colas con la información almacenada en el archivo de configuración. A partir de esta información podrán realizarse cambios en el esquema actual. De no contarse con un archivo de configuración inicial, se deberán cargar todos los datos.

3) int menu_ppal( )

Es una función utilizada para presentar menúes en pantalla. Las funciones soportadas por el menú principal son:

1. Especificar una cola de rechazo.
2. Altas, bajas y modificaciones de Colas.
3. Altas, bajas y modificaciones de Procesos.
4. Salir.

Las funciones que llevan a cabo estas acciones son:

i) int in_colarechazo( )

Esta función se encarga de actualizar la cola de rechazo. Verifica que la cola especificada exista en el sistema.

ii) int menu_ABM( )

Esta función se encarga de presentar el menú de altas, bajas y modificaciones de colas y procesos.

Colas

La función que realiza el alta de una cola se encarga de inicializar la tabla de colas para las colas nuevas con los datos necesarios. Estos son el nombre de la cola, el nombre del directorio donde se encuentran los datos de la misma, así como otros valores necesarios para la ejecución de servidores RPC que están asociados a las colas. Mantiene un contador con la cantidad de colas existentes en el sistema. Este valor se escribirá en el archivo de configuración.

La función que elimina colas del esquema debe verificar que la cola a eliminar no esté siendo utilizada. La cola a borrar no debe ser cola de entrada o salida para ningún tipo de proceso existente. Si se deseara eliminar una cola asociada a algún tipo de proceso

se debería modificar el tipo de proceso para que no la use y luego eliminar la cola del esquema.

La función de modificación de colas permite alterar el directorio donde se almacenan los datos contenidos en dicha cola. Esta es la única propiedad de una cola que el operador puede desear cambiar ya que las restantes se inicializan en forma automática y se refieren a características de implementación de los servidores de RPC asociados a las colas.

Procesos

La función que realiza el alta de un proceso se encarga de inicializar la tabla de procesos con datos como ser el nombre del proceso. Mantiene un contador con la cantidad de procesos existentes en el sistema. Este valor se escribirá en el archivo de configuración. En la versión interactiva, el operador debe proporcionar la información referida a las colas de entrada y salida para cada tipo de proceso. Por este motivo la opción de alta de tipos de procesos permite la especificación de las colas de entrada y salida para el tipo de proceso a ingresar. Para cada cola, se verifica que exista en el sistema.

Se presenta una función que elimina tipos de procesos.

La función de modificación de tipos de procesos permite cambiar las colas de entrada y salida de un tipo de procesos. Se permite agregar y eliminar nuevas colas de entrada y salida para el tipo de proceso. Para cada caso se verifica que la operación se realice sobre colas existentes en el esquema.

4) int salvar_config( )

Esta función escribe un archivo de configuración que será usado por los managers para inicializar el esquema de procesos y colas. El formato del mismo es fijo para ser fácil de reconocer por el *connectserver* que inicializa las tablas del sistema y determina la cantidad de servidores a ejecutar con la información almacenada en el archivo de configuración. Es la misma función utilizada en la versión anterior del configurador.

6.2 Implementación de las operaciones a realizar por los clientes.

Por lo visto en la introducción al esquema de los procesos que conforman los managers y las operaciones descriptas en el lenguaje, se deberán soportar las operaciones introducidas en la sección 5.1.2 "Clientes de la herramienta", página 44.

A continuación se presentan las funciones que permiten a los clientes hacer uso de las operaciones soportadas por la herramienta. Estas funciones constituyen las APIs que la herramienta provee a los clientes.

```
rc = Present(char * tipoproc, IdType * id);
```

Antes de comenzar la ejecución del proceso cliente, se ejecuta esta función para que el servidor realice un alta en su tabla de procesos clientes. Se especifica el tipo de proceso que utilizará el cliente. Este determina las colas con las que se puede conectar el cliente y operaciones puede realizar sobre las mismas.

Como parámetro de salida se recibe un identificador único, que el proceso tendrá que utilizar cada vez que quiera comunicarse con los managers.

```
tipoproc:   Nombre del tipo de proceso que se desea conectar.
            Parámetro de Entrada.
id      :   Handle del cliente. Parámetro de Salida.
rc      :   Entero que contiene el resultado de la
            operación. Código de retorno de la función.
```

```
colaPtr = ConectarCola(IdType id, char * nomcola, char
  *dir);
```

Esta función inicializa los recursos y hace accesible la cola al cliente. El primer parámetro es el identificador de cliente (id) devuelto por la función Present. El segundo parámetro es un string que especifica el nombre de la cola con la que se quiere conectar. El tercer parámetro retorna el nombre del directorio (path completo) asociado a la cola. De este directorio se leerán los datos si es usada como cola de entrada o se escribirán si es usada como cola de salida.

Retorna un puntero o handle a la cola con la que se desea conectar. Se utilizará este identificador de la cola en futuras llamadas al manager. Se utiliza un identificador para que el servidor, en subsecuentes requerimientos pueda acceder directamente a la cola y no deba realizar una búsqueda por el nombre de la cola. Si hubo error retorna NULL.

```
id      : IdType que identifica al cliente. Parámetro de
          Entrada.
nomcola : Puntero a carácter donde se encuentra el nombre de
          la cola con la que se desea conectar. Parámetro de
          Entrada.
```

```
dir     : Cadena de caracteres que contiene el directorio
          (path completo) asociado a la cola. Esto contiene
          el path completo sin la barra del final. De allí
          se levantarán los datos si es usada como cola de
          entrada o se escribirán si es usada como de
          salida. Parámetro de salida.
colaPtr: Puntero o handle de la cola. Es de tipo Cola.
```

`rc = CrearToken(Cola *colaptr, char * nomtoken);`

Esta función se utiliza para crear nuevos trabajos. Implementa la operación *crear* analizada anteriormente. *Crear* introduce por primera vez un trabajo al flujo del sistema. Tiene como parámetros el handle de la cola en la que se desea agregar el trabajo y el nombre del token a crear. Devuelve un código de error de tipo entero.

```
colaptr: Puntero a Cola que indica la cola a la que se
         desea agregar un trabajo nuevo para el sistema.
         Parámetro de Entrada.
nomtoken: String donde se especifica nombre del trabajo a
         crear. Parámetro de Entrada.
rc      : Entero que contiene el resultado de la operación.
```

`rc = TomarToken(IdType id, Cola *colaptr, char * nomtoken);`

Esta función solicita un nuevo trabajo a los servidores. Es la función que implementa la operación *tomar* analizada anteriormente. Tiene como parámetros al identificador (*id*) que fue devuelto por la función Present, el handle de la cola de la que desea obtener un trabajo y el nombre del trabajo a tomar. Si el tercer parámetro es vacío, significando que no se pide ningún trabajo en particular, pide el primer trabajo libre y retorna el nombre del token otorgado en el mismo parámetro. Si se especifica el niombre de un trabajo, toma el token especificado por nomtoken. Devuelve un código de error de tipo entero.

```
id      : IdType que identifica al cliente. Parámetro de
          Entrada.
Colaptr: Cola que indica la cola de la que se desea obtener
          un trabajo. Parámetro de Entrada.
Nomtoken:Puntero a carácter donde la función lee/retorna el
          nombre del trabajo. Parámetro de Entrada/Salida.
rc      : Entero que contiene el resultado de la operación.
```

`rc = PasarToken(IdType id, Cola *colaorigen, Cola *coladest, char *nomtoken);`

Esta función notifica que se terminó de trabajar con el trabajo *nomtoken* y lo pasa de la cola *colaorigen* a la *coladest*. Es la función que implementa la operación *poner*

analizada anteriormente que habilita a los managers a entregar ese trabajo a algún proceso que lo solicite en la cola *coladest*.

Tiene como parámetros el identificador (*id*) que fue devuelto por la función Present, el handle de la cola a la que pertenece actualmente el trabajo, el handle de la cola a la que se desea pasar el trabajo ya terminado y el nombre del trabajo. Devuelve un código de error de tipo entero.

```
id       : IdType que identifica al cliente. Parámetro de
           Entrada.
colaorigen: Puntero a Cola que indica la cola a la que
           pertenece el trabajo. Parámetro de Entrada.
coladest: Puntero a Cola que indica la cola a la que se
           desea pasar el trabajo. Parámetro de Entrada.
nomtoken: Puntero a carácter donde se especifica el nombre
           de trabajo que se desea pasar. Parámetro de
           Entrada.
rc       : Entero que contiene el resultado de la operación.
```

`rc = BorrarToken(IdType id, Cola *colaptr, char * nomtoken);`

Esta función elimina un trabajo del flujo del sistema. Es la función que implementa la operación *borrar* analizada anteriormente. Tiene como parámetros al identificador (*id*) que fue devuelto por la función Present, el handle de la cola a la que pertenece el trabajo y el nombre del token a borrar. Devuelve un código de error de tipo entero.

```
id       : IdType que identifica al cliente. Parámetro de
           Entrada.
colaptr: Cola que indica la cola a la que pertenece el
           trabajo. Parámetro de Entrada.
nomtoken: Puntero a carácter que especifica el nombre del
           trabajo. Parámetro de Entrada.
rc       : Entero que contiene el resultado de la operación.
```

6.3 Implementación de los servidores que conforman los managers

En la presente sección se describirán a alto nivel detalles de implementación de los servidores que componen los managers y los mecanismos que utilizan para su interacción.

6.3.1 Implementación de las estructuras comunes y mecanismos de comunicación

6.3.1.1 Estructuras de datos

Como se introdujo en la sección 5.2.1.1 " Diseño de las estructuras de datos", en la página 46, se implementan las siguientes tablas:

- tabla de colas,
- tabla de trabajos,
- tabla de clientes,
- tabla de procesos.

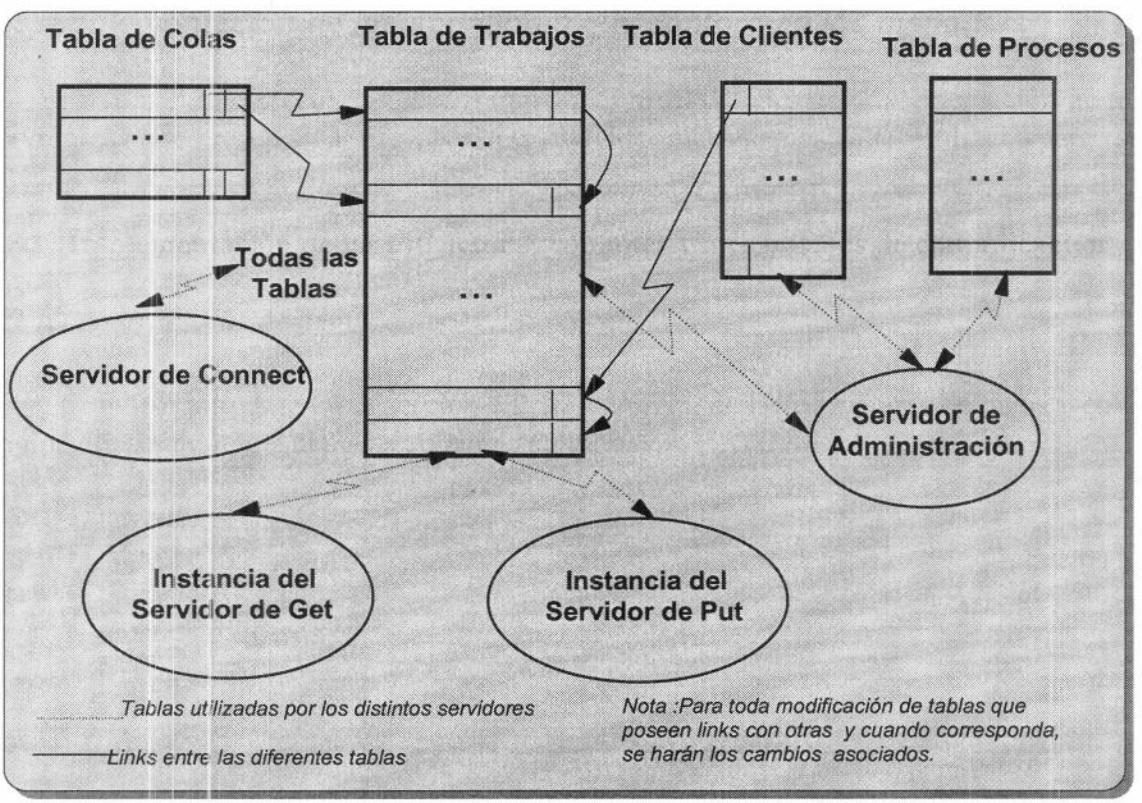


Figura 8 : Tablas de estructura de datos en memoria

Tabla de Colas

La Tabla de Colas posee información asociada a cada cola definida en el sistema, con datos como el nombre de la cola, el identificador dentro del sistema, la cantidad de trabajos encolados en la misma, un semáforo binario de exclusión mutua y los vínculos mencionados previamente.

- El nombre de la cola es la forma que inicialmente tiene un cliente de la herramienta para identificar la cola sobre la cual desea operar.
- El identificador es utilizado internamente para identificar las colas de una manera más eficiente que mediante el nombre.
- La cantidad de trabajos encolados se mantiene para poder implementar semáforos contadores infinitos. Esta decisión se tomó a raíz de una limitación en la implementación de semáforos en Unix, cuyo valor máximo soportado es de 35565. Para permitir la existencia de más trabajos en una cola, se implementó un contador y se utilizó un semáforo binario para proteger el acceso a la cola y modificación del valor del contador que almacena la cantidad de trabajos encolados, evitando así posibles inconsistencias debido al acceso concurrente de diferentes servidores.
- Existen dos vínculos hacia la tabla de Trabajos, uno hacia el primer trabajo de la cola y otro hacia el último trabajo. Existen por razones de eficiencia y para evitar recorrer todos los trabajos de la cola para encolar o para tomar el primer trabajo.

Tabla de Trabajos

En la tabla de trabajos se almacenan los datos de los trabajos que administra el sistema. Es el repositorio de la totalidad de los trabajos ubicados en las diferentes colas. Las entradas en la tabla que no están utilizadas se marcan como **freeslot.** La tabla de trabajos contiene:

- el nombre del trabajo,
- el identificador de la cola donde se halla el trabajo (FREE cuando es un freeslot),
- la cantidad de veces que el trabajo provocó la muerte del cliente que lo estaba procesando.

Tabla de Procesos

En esta tabla es donde se almacenan los diferentes Tipos de Procesos que el sistema soporta. Esta tabla será consultada cada vez que un cliente se presente a los managers y cada vez que solicite conectarse a una cola. Se mantiene información de:

- las colas que cada tipo de proceso puede acceder,
- autorización para conectarse
- operaciones pueden realizar sobre las mismas.

Tabla de Clientes

En esta tabla se almacena toda la información relacionada a los clientes que se encuentran utilizando los servicios provistos por los managers. Para agregar un nuevo cliente en la tabla es necesario conocer el Tipo de Proceso del mismo. Este dato es utilizado por la herramienta para verificar los permisos que posee sobre las diferentes colas del sistema, y el tipo de operaciones que puede realizar sobre las mismas.

Pueden existir varias instancias de clientes de igual Tipo de Proceso. Para poder identificarlas unívocamente, cada entrada en la tabla mantiene actualizado un identificador que es retornado al cliente. El cliente usará este identificador cuando solicite un servicio a los managers. Cuando un cliente se cae o termina su procesamiento, provoca que se recicle el identificador que tenía asociado.

Cuando un cliente toma un trabajo, dicho trabajo pasa a un estado de retenido por el cliente en la cola. El cliente tiene un vínculo hacia la tabla de trabajos, manteniendo un encadenamiento de todos los trabajos que tiene retenidos y que está procesando. No se diseñaron dos vínculos, uno para el primer trabajo retenido y otro para el último, debido a que la cantidad de trabajos que puede estar procesando simultáneamente un cliente no será tan grande. Por otro lado, al buscar un trabajo en particular para Pasar o Borrar, se deben revisar todos los trabajos retenidos por el cliente.

La tabla contiene:

- el identificador del cliente,
- el tipo de proceso del cliente,
- el encadenamiento a la tabla de trabajos.

6.3.1.2 Implementación del mecanismo de protección de recursos

En la presente sección se describen detalles de implementación del mecanismo de protección de los recursos compartidos y se muestran las funciones que implementan el mecanismo descripto. En la sección 5.2.1.2 "Diseño del mecanismo de protección de recursos", página 50 se especificó el diseño de este mecanismo.

Para proteger a las colas se utiliza un semáforo por cola, donde cola se refiere al concepto de cola como almacenamiento de trabajos. En lo subsiguiente al decir cola nos referimos a tanto a los trabajos libres para esa cola -los que aún no han sido tomados por ningún cliente-, como los trabajos que están siendo procesados por clientes. Desde el punto de vista de las tablas, una cola en este sentido es la entrada de la misma en la tabla

de colas, el encadenamiento de los trabajos libres pertenecientes a dicha cola que son almacenados en la tabla de trabajos y el encadenamiento de los trabajos apuntados desde la tabla de clientes en esa cola.

La estructura donde se almacenan los trabajos está organizada en registros o slots. La misma tiene slots vacíos que servirán para guardar los datos de los nuevos trabajos que se vayan creando en el sistema. El conjunto de slots libres de la estructura se maneja como una cola más a la que llamaremos ***FreeSlot.*** Se mantiene un semáforo para proteger esta cola también.

En la implementación del mecanismo de protección, se utilizó un conjunto de semáforos de Unix para garantizar exclusión mutua en el acceso a las colas. Las mismas son los recursos que almacenan los datos de los trabajos.

Deben protegerse diferentes colas dependiendo de la operación a realizar. En esta sección ampliaremos el esquema visto en "Utilización de semáforos para lograr exclusión mutua".

Debido al mecanismo de encadenamiento de los trabajos en las estructuras de datos, para algunas operaciones es necesario bloquear a todas las colas de entrada al tipo de proceso que solicita la operación. No es suficiente bloquear sólo la cola sobre la cual se realizará la operación

6.3.1.2.1 Acceso al conjunto de semáforos

Se utiliza una única función para acceder al conjunto de semáforos.

ArmarVecColas(Tipo_de_Proceso, Tipo_de_Función, Cant, ColaDest, ColaRechazados, SemVec)

donde:

Tipo_de_Proceso es un índice al tipo de proceso almacenado en la tabla de procesos. Este parámetro indica cuáles son las colas de entrada y salida del proceso que solicita la operación.

Tipo_de_Funcion indica la operación a realizar, pudiendo ser CrearToken, TomarToken, BuscarToken, PasarToken, BorrarToken o ClientDown. Dependiendo del tipo de operación que se quiere realizar, se pedirá exclusión mutua a las distintas colas que intervengan.

Cant es un parámetro de salida que retornará la cantidad real de colas que fueron bloqueadas.

ColaDest Las operaciones que deben mover un trabajo de una cola a otra, necesitan indicar cuál es la destino para conseguir exclusión mutua sobre la misma.

ColaRechazados Para la operación de ClientDown, en el caso que un trabajo haya excedido la máxima cantidad de rechazos permitidos, éste no vuelve a su cola original,

sino a una cola especialmente designada para recibir aquellos trabajos que provoquen varios errores sobre las aplicaciones.

SemVec El identificador del conjunto de semáforos.

6.3.1.2.2 Implementación de las operaciones: uso de semáforos.

A continuación se detalla la utilización de la función ArmarVecColas para cada una de las operaciones que realizan los clientes de los managers.

Operación CrearToken

```
ArmarVecColas( -- , CREAR, &cant, cola_dest, -- , SemVec)
```

No es necesario indicarle el tipo de proceso ya que al crear sólo se pide exclusión mutua sobre la cola destino especificada en cola_dest que es la cola donde se quieren crear el trabajo. Debido a que se debe tomar un slot libre para agregar el nuevo trabajo se solicita exclusión mutua sobre la cola FREE, que es la que mantiene el encadenamiento de slots libres. En la variable cant retorna la cantidad de colas que se bloquearon para obtener exclusión mutua. No es necesario pasarle cuál es la cola de rechazados.

Operación PasarToken

```
ArmarVecColas( TipoProc, PASAR, &cant, cola_dest, --, SemVec)
```

Es necesario indicar el tipo de proceso, ya que se obtendrá la información relativa a qué colas de entrada hay que bloquear. También debe bloquearse la cola_dest. Así se logra que un proceso pueda tomar trabajos de más de una cola de entrada. No es necesario indicar la cola de rechazados.

Operaciones BorrarToken, TomarToken y BuscarToken

```
ArmarVecColas( tipoProc , BORRAR/TOMAR/BUSCAR, &cant, -- , -- , SemVec)
```

Esta llamada es similar a la anterior, aunque no se utiliza la cola destino. Tampoco hay que indicar la cola de rechazados. Para el caso de BorrarToken, hay que bloquear la cola FREE ya que se devolverá un slot a la tabla.

ClientDown

ArmarVecColas(TipoProc ,CLDOWN, &cant, -- , cola_rechazados SemVec)

Esta llamada se va a realizar cuando se produzca un ClientDown. Significa que un cliente que estaba ejecutando murió por alguna razón, y se debe proceder a liberar todos los trabajos que tenía tomados. Cada trabajo debe retornar a su cola de entrada. Se necesita exclusión mutua en todas la colas de entrada del proceso, información que se obtiene a través del parámetro TipoProc. Asimismo, si un trabajo fue rechazado más de *n* veces en lugar de volver a su cola original, va a ir a la cola de rechazados especificada en el quinto parámetro.

6.4 Implementación del servidor de Connect: connectserver

En la presente sección se describen detalles de implementación del servidor connectserver y las funciones que lo componen que fueron diseñados en la sección 5.3 "Diseño del servidor de Connect: Connectserver", página 55.

El *connectserver* es el padre de todos los demás servidores: *getserver, putserver, administrador, historyserver, logserver y monitor*. Es el encargado de inicializar todas las estructuras comunes a los servidores, como ser tablas de memoria compartida, semáforos y colas de mensajes. Crea los procesos que luego conformarán la estructura de los *managers*. Luego se presenta como un servidor de RPC para recibir solicitudes de conexión a las colas por parte de los clientes.

A partir de la información proporcionada por el esquema de colas calcula la cantidad de instancias de cada uno de los servidores a crear, así como el tamaño de las estructuras para almacenar los trabajos.

6.4.1 Detalle de las funciones del Connectserver.

En la presente sección se presentan las acciones realizadas por cada una de la principales funciones del connectserver y su implementación.

Inicializar

La función Inicializar se ejecuta antes que el connectserver se transforme en servidor de RPC. Esta función es la responsable de crear el ambiente de ejecución de los restantes procesos integrantes de los managers, así como de inicializar las estructuras de datos.

Las acciones que realiza la función Inicializar son:

1. Realiza las acciones necesarias para que el connectserver sea un demonio de UNIX y se asegura que exista una única instancia del servidor ejecutando.
2. Setea las señales para especificar las acciones a tomar en caso de muerte normal o anormal del connectserver y de cualquiera de los servidores hijos del mismo.
3. Setea el umask para establecer los permisos de ejecución correctos.
4. Inicializa el syslog.
5. Lee la configuración para el esquema actual.
6. Inicializa las tablas del sistema. Estas son las estructuras de memoria compartida utilizadas por todos los servidores.
7. Inicializa los semáforos compartidos por todos los servidores.
8. Crea las instancias de cada uno de los servidores para el esquema actual.

9. Almacena los nombres de los recursos del sistema que están siendo utilizados por la ejecución actual de los managers. Con esta información se liberan automáticamente los recursos en caso de falla o muerte de los managers.

Las acciones arriba descriptas son implementadas de la siguiente manera:

1. Esta acción es realizada por una función de biblioteca **InicializarProceso(ArchivoDeLock)** que se halla en libdaemon.a. Realiza las acciones necesarias para convertir al proceso que la ejecuta en un demonio.

2. Buscamos una forma elegante y efectiva para finalizar la ejecución de un conjunto de servidores. Es fundamental que si alguno de los servidores sufre algún inconveniente y deja de funcionar, ninguno de los demás siga haciéndolo ya que podría provocar inconsistencias en las tablas de trabajo. Existe una instancia de servidor ejecutando para los distintos tipos de operaciones sobre cada una de las colas, por lo tanto, de finalizar algún servidor no se podría seguir operando sobre la cola asociada, pero sí sobre las demás. Se ha decido que lo más conveniente es terminar al conjunto de los managers en caso de falla.

   Esta funcionalidad sólo debería utilizarse para parar en forma controlada la ejecución de los managers. Un administrador con el nivel de permisos adecuado debería correr una utilidad especial para finalizar la tarea. Se ha logrado que al matar a cualquiera de los servidores se consiga la finalización ordenada y en cascada de todos los servidores acompañada por la liberación de los recursos utilizados.

   Se realizan las acciones necesarias para que en caso de finalización, interrupción o muerte de alguno de los servidores integrantes de los managers se limpien todos los recursos utilizados. Se logra este objetivo seteando las señales SIGTERM, SIGINT y SIGCHLD para que ejecuten la función **Salir** y especificando que en caso de finalizar el proceso se ejecute la función **LimpiarRecursos**.

   La función **Salir** es ejecutada cuando el connectserver es matado por alguna señal que viene del exterior del proceso o bien cuando alguno de sus hijo muere. La función **Salir** provoca un exit del proceso. De esta forma se realizan las mismas acciones tanto en caso de una finalización provocada desde el exterior como si el *connectserver* realizara un exit. La ventaja es que se controla la situación exactamente de igual manera en caso de muerte normal o anormal del proceso.

   La función **LimpiarRecursos** se encarga de liberar todos los recursos utilizados por el conjunto de servidores que conforman los managers y de matar a todos ellos.

3. Se setean los permisos que serán utilizados por el sistema al crear archivos y realizar operaciones de entrada/salida. Se garantiza que los recursos del sistema creados por los managers posean los permisos adecuados para su ejecución y

seguridad provistos por el sistema operativo UNIX. Se setea el umask en 0 (cero) para que sean tenidos en cuenta los permisos especificados por las operaciones de creación.

4. Se inicializa el syslog que luego será utilizado por todos los servidores que conforman los managers para hacer logging de sus acciones.
   El syslog es un servicio que brinda el sistema operativo UNIX. Existe un demonio que presta el servicio de hacer el logging de todas las cadenas de caracteres que se le pasen como parámetro.

5. Se almacena dentro de un archivo, la configuración de la arquitectura actual. Se especifican las colas, los tipos de proceso, la cola de rechazo designada y las operaciones permitidas sobre cada una de las colas dependiendo de los tipos de proceso. Por cada tipo de proceso se detallan las colas de entrada, salida e iniciales. Sobre las colas de entrada puede hacerse una operación TomarToken o BuscarToken, sobre las colas de salida puede hacerse un PasarToken y BorrarToken. Sobre las colas iniciales se puede ejecutar un CrearToken.

   Las razones para dividir las operaciones de esta manera quedarán más claras al estudiarse las propiedades de cada uno de los servidores que ejecutan las operacions mencionadas.

   La siguiente función realiza parte de las acciones antes mencionadas. El resto de la información se lee del archivo de configuración en el momento más indicado para su uso.

   int cantcolas_config(int *cantproct, char *cola)

   Lee del archivo "config.server" la cantidad de colas definidas y la guarda en **cantcolas** (variable global). La cantidad de tipos de procesos definidos lo guarda en **cantproct** (parámetro) y el nombre de la cola de rechazos es almacenado en **cola** (parámetro).

6. Se inicializan las estructuras de memoria que almacenarán durante la ejecución de los managers toda la información relativa a los trabajos que estos distribuyen.

   Las estructuras manejadas, así como su interrelación han sido explicadas más arriba en detalle. Se inicializan las tablas de colas, de trabajos, de clientes, de tipos de procesos y la cola de mensajes para el logging interno de trabajos. Se deben inicializar las tablas con los valores que correspondan, en el caso de la tabla de trabajos, deben recuperarse los trabajos de la corrida anterior de los managers. Aunque el esquema de logging utilizado es explicado más adelante debido a su complejidad y extensión, adelantamos que se utilizan las rutinas de recuperación

que se encargan de levantar la información almacenada en los medios persistentes que manejan los managers.

Cada una de las tablas se inicializa con los valores adecuados en la función **int inic_tablas(int cantproct):**

Tabla de Colas: se leen del archivo de configuración los nombres de cada una de las colas. Cada una posee un contador indicando cuantos trabajos hay en la misma y un semáforo binario de exclusión mutua sobre la variable del contador. Estos son inicializados en el siguiente punto (inicialización de semáforos de los managers).

Tabla de trabajos: se inicializan con los trabajos de la corrida anterior.

Tabla de clientes: se inicializa cada registro de esta estructura con un número identificatorio que se mantendrá único a través de la ejecución de los manages para cada uno de los clientes que hagan uso de sus servicios. Al inicializarse la tabla se da a cada registro su número de orden (índice) como identificador del cliente. El esquema de preservación de identificación único se tratará más adelante cuando se estudie la modificación de la tabla de clientes.

7. Inicializa los semáforos compartidos por todos los servidores. Los semáforos de Unix se utilizan para tres funciones diferentes en los managers.

   La primera de ellas es un semáforo de exclusión mutua sobre cada cola para el acceso a las operaciones que proveen los managers. Como se explicó, es necesario proteger las colas porque ciertas operaciones podrían realizarse concurrentemente por diferentes instancias de servidores. Hay un único semáforo de UNIX con un valor para cada una de las colas, que asegura la exclusión mutua en forma atómica sobre las colas especificadas. Ver sección ***Protección de los recursos durante la ejecución***.

   Otra función más común es la de semáforo de exclusión mutua individual sobre la variable contadora de la cantidad de trabajos pendientes en cada cola. No se usa un semáforo contador directamente para almacenar la cantidad de trabajos dada la limitación de los semáforos de UNIX que permiten un máximo de 35565 como valor. En este esquema usamos una variable que pueda almacenar números tan grandes como queramos y un semáforo binario que garantice exclusión mutua en el acceso a la misma. Este es necesario para que dos instancias de servidores diferentes no modifiquen la variable a la vez y quede en un estado inconsistente en referencia a la cantidad de trabajos realmente existentes en la cola.

   El tercer tipo de semáforos es de exclusión mutua sobre el esquema de logging. Existe un semáforo por cada uno de los archivos buffer de logging manejados. Se

explica en detalle en la sección Diseño e Implementación del servidor de Historia: HistorySever.

La función usada es:

int inic_sem()

8. Crea cada uno de los procesos servidores que conforman el esquema de los managers.

9. Almacena en un archivo de logging los nombres de los recursos del sistema que están siendo utilizados por la ejecución actual de los managers (memoria compartida, semáforos, colas de mansajes, identificadores de cada uno de los procesos que conforman los managers).

ConectarCola

La función ConectarCola es la función exportada por el connectserver una vez que se ha transformado en servidor de RPC. Los clientes de la herramienta solamente pueden acceder a esta función del Connectserver. Al ser un servidor de RPC el connectserver ejecuta un ciclo infinito esperando el pedido de servicio que los clientes solicitan a través de ConectarCola. Este módulo es el encargado de manejar la seguridad y cualquier requerimiento asociado a control se desarrollará en este módulo.

Las acciones que realiza la función ConectarCola son:

1. Verifica la validez del identificador del cliente.
2. Según los permisos de acceso del cliente retorna los identificadores de la cola adecuados para el mismo opere.

A continuación se presenta la función ConectarCola y sus parámetros, así como el detalle de implementación de las acciones arriba descriptas.

void ConectarCola (int ClienteId, char* NombreCola, int* ColaId, u_long* putProgNumber, u_long* getprogNumber, char* Pathname)

No todos estos parámetros son utilizados por el programador de la aplicación cliente, pero si son imprescindibles para que el cliente pueda acceder a los servidores encargados de operar con la cola cuyo nombre se especifica en NombreCola.

1. El ClienteId es utilizado por el connectserver para verificar la validez del identificador del cliente.

descriptor = (descriptor + cant_entradas) módulo valor_máximo

Este valor incrementado se transforma en el nuevo descriptor de la entrada cuando sea alocada la próxima vez. Los procesos que traten de acceder a la entrada por su descriptor viejo, fallan en el acceso y reciben un error.

2. Se retorna el identificador de la cola NombreCola en la variable ColaId y el directorio donde se encuentran los datos de los trabajos de esa cola en PathName. De acuerdo a los permisos de acceso del cliente a la cola se retornan los identificadores de los servidores adecuados. Según el tipo de proceso del cliente se sabe si la cola especificada es de entrada, salida o inicial. En caso de ser una cola de entrada se retorna el identificador de los servidores de get y put para esa cola. Si sólo fuera una cola inicial se retorna el de put, y si fuera de salida solo se le permite conectarse. Esta separación se debe principalmente a la implementación a nivel RPC de los servidores que conforman los managers.

El stub de los clientes luego utilizará el identificador de servidor retornado para efectuar operaciones directamente con los servidores de la cola especificada. La aplicación cliente sólo sabe que está realizando una operación, sin conocer el detalle de los parámetros.

El connectserver actua como un servidor de ports (portmapper) que indica a las aplicaciones la dirección -en este caso de RPC- donde se encuentran los servicios requeridos. Esto sólo se realiza una única vez por ejecución. Los clientes, una vez conectados a la cola, seguirán usando el identificador devuelto sin necesidad de pasar por el connectserver nuevamente.

6.5 Implementación del servidor de get: getserver

En la presente sección se describen detalles de implementación del servidor getserver y las funciones que lo componen, que fueron diseñados en la sección 58 "Diseño del servidor de get: Getserver", página 58.

Este servidor es el encargado de recibir los pedidos de trabajos que realizan los clientes que usan las APIs de la herramienta y entregar trabajos a los clientes que así lo requieran. Brinda dos clases de servicios, uno bloqueante y otro no bloqueante.

Controla la correcta administración de los trabajos asegurando la consistencia en el flujo de los mismos entre las colas y los requerimientos de los clientes. Se asocia una instancia de servidor a cada cola que exista en el esquema.

El getserver está implementado para administrar una cola FIFO, el primer trabajo que se deposita es el primer trabajo en ser entregado ante un requerimiento. La administración de la cola se definió en un módulo independiente y es fácilmente ampliable a soportar cualquier otro esquema de ordenamiento o prioridades para la entrega de trabajos. Una variedad de servicio provisto es la entrega de un trabajo específico solicitado por un cliente.

6.5.1.1 Detalle de las funciones del Getserver.

Este módulo consta de una función de inicialización y dos funciones principales. Las funciones principales se diseñaron para soportar el servicio bloqueante y no bloqueante de este servidor.

1) TomarToken (Id Cliente, Cola) -> Trabajo

La función asociada al servicio bloqueante es TomarToken que recibe como parámetros la identificación del cliente que está haciendo el requerimiento y el identificador de la cola de donde se desea tomar el trabajo. El valor retornado es el identificador del trabajo. Si no se puede satisfacer el requerimiento, el servidor se bloquea, bloqueando en consecuencia al cliente hasta que sean depositados nuevos trabajos en esa cola.

Para soportar el comportamiento bloqueante se diseñó un esquema de semáforos asociados al acceso a todo recurso compartido que lo necesitase. Para mayores detalles relativos al manejo de los semáforos referirse al Anexo A.

Se utiliza un semáforo para bloquear al servidor en caso de no haber más trabajos disponibles. Se ejecuta un P(X) sobre el semáforo asociado a la cola de donde se quiere tomar el trabajo. El servidor se bloqueará siempre que no haya nada que tomar, caso contrario entrega el primer trabajo de la cola. Se evita tomar otros recursos sin tener la

seguridad que hay trabajos para entregar y correr el riesgo de paralizar el funcionamiento de todo el sistema. Una vez validada la existencia de trabajos, se consigue la exclusión mutua para acceder a todas aquellas estructuras compartidas entre las diferentes instancias de servidores que componen el sistema. Cuando se obtiene acceso a los diferentes recursos se procede a desencolar el trabajo y asociárselo a la lista de trabajos del cliente -esta lista contiene todos los trabajos que fueron entregados a dicho cliente y que todavía no ha terminado de procesar-. De esta forma queda registrado que este determinado trabajo está siendo procesado por el cliente y el trabajo pasa a estar en un estado de retenido, sin haber sido eliminado de la cola. Los módulos responsables de eliminarlo se describen en el putserver. Una vez ubicado el trabajo en las listas correspondientes, se registra que se ha consumido un trabajo de la cola X, decrementando en 1 la cantidad de trabajos encolados en espera de su procesamiento. No se considera el estado de retenido como estado disponible de consumir, por lo tanto se decremeta el valor de los trabajos a entregar.

Una vez que las variables reflejan el cambio de estado del trabajo entregado, se liberan los recursos que tenía tomados y se retornar al cliente el trabajo que se le asignó.

2) BuscarToken (Id Cliente, Cola, Trabajo) -> Trabajo

Para la modelización del servicio no bloqueante se diseñó la función BuscarToken, que recibe la misma información que TomarToken más el nombre del trabajo que se requiere. Esta función retorna un código indicando si tuvo éxito la búsqueda del trabajo o no. Si la función recibe como nombre del trabajo requerido el valor nulo, retorna el primer trabajo disponible, siendo ésta la alternativa no bloqueante a la función TomarToken.

Ambas funciones se comportan de forma similar una vez identificado el trabajo a asignar. La diferencia radica en que esta función no se bloquea en un semáforo en caso de no exitir trabajos. Se consulta la variable asociada a la longitud de la cola de trabajos encolados en la cola pedida, si este valor es mayor a cero, se solicita exclusión mutua para el acceso a los recursos compartidos y se procede a buscar el trabajo pedido por el cliente. Si la búsqueda no es exitosa, se retorna un código de error que refleje este resultado. Sino, se procede de la misma manera que en la función TomarToken antes descripta. Si no hay trabajos en la cola, se informa de la situación.

3) ParseArgs (*ID estructuras compartidas*)

La tercer función de este servidor de trabajos, se encarga de la inicialización del entorno para la correcta ejecución de este servidor en sincronización con los demás servidores de la herramienta.

El punto de entrada para el getserver es la función ParseArgs. Esta función se ejecuta cuando comienza el servidor y se encarga de parsear los argumentos que le pasa el connectserver al getserver en el momento de realizar el fork del mismo. El getserver obtiene los identificadores de las tablas en memoria compartida y del conjunto de semáforos que utilizará a través de estos parámetros. Esta función inicia el logging en el syslog.

Una vez ejecutada la función ParseArgs, el getserver se transforma en un servidor que provee los servicios implementados por las funciones TomarToken y BuscarToken.

6.6 Implementación del servidor de put: putserver

En la presente sección se describen detalles de implementación del servidor putserver y las funciones que lo componen como se diseñaron en la sección 5.5 "Diseño del servidor de put: Putserver", página 62.

Este servidor es el encargado de recibir los pedidos para agregar nuevos trabajos a la cola que está administrando. Las funciones soportadas por este módulo son no bloqueantes. No existen servicios que mantengan bloqueado tanto al cliente como al servidor, exceptuando los bloqueos temporales consecuencia de esquemas de acceso a regiones críticas.

El putserver brinda todos los servicios relacionados a la manipulación de trabajos excluyendo la entrega de los mismos, que es responsabilidad del getserver. Las funciones de este servidor proveen los servicios de agregar un trabajo a una determinada cola, poner un trabajo en una cola inmediatamente después de haber sido creado, introduciendo un trabajo nuevo al flujo del sistema y borrar un trabajo, eliminándolo del flujo total.

Todas las operaciones que realice este módulo impactan directamente en un cambio del estado del trabajo. Estos cambios se registran a nivel de memoria, por lo tanto una falla implicaría perder el estado general en que se encontraba el sistema. Por este motivo, se decidió utilizar un mecanismo de logging en combinación con un mecanismo de manejo de historia, los cuales se explican con más detalle en las secciones de servidor de logging y servidor de historia. Se registra cada movimiento o cambio de estado permitiendo llevar una imagen del estado del sistema y posibilitar la reconstrucción de las colas al estado anterior a producirse la falla.

El putserver es un servidor de RPC.

6.6.1.1 Detalle de las funciones del Putserver

1) CrearToken (Cola, Trabajo) -> Ok?

Un trabajo se crea cuando un cliente, que no tenía ningún trabajo retenido para transformar, genera con sus propios datos un nuevo trabajo y lo ingresa al sistema agregándolo en alguna de las colas.

La función asociada al servicio de crear un trabajo es CrearToken que recibe como parámetro la cola donde se debe encolar el nuevo trabajo y la identificación del mismo.

Esta función ejecuta las rutinas asociadas al ingreso a la región crítica, tomando para uso exclusivo aquellas estructuras de datos que intervienen en la creación de un

trabajo. Una vez garantizada la consistencia de la operación evitando accesos concurrentes, se agrega un nuevo trabajo en una cola. Un nuevo ítem que se va a asociar a este trabajo se agregar a la estructura que modeliza una cola y se incrementa en uno la cantidad de trabajos disponibles para ser consumidos.

Debe registrarse en el logging la introducción de un trabajo nuevo al sistema y cuál es la cola destino de este trabajo. La función WriteLog() es la encargada de implementar la comunicación con el servidor de Logging. De esta manera se garantiza el registro del ingreso al sistema.

2) BorrarToken (Id Cliente, Cola, Trabajo) -> Ok?

Un trabajo se *borra* cuando el cliente que lo tiene retenido solicita la eliminación del trabajo del flujo del sistema. Al ser borrado, se elimina de la cola donde se encontraba retenido. Cada vez que un cliente tiene retenido un trabajo para su transformación siendo ésta la última transformación necesaria para generar un resultado, se está en condiciones de eliminarlo.

La función asociada al servicio de borrar un trabajo es BorrarToken. Recibe como parámetros el identificador del cliente que lo tenía tomado, el identificador del trabajo que se desea borrar y la cola donde se encuentra retenido dicho trabajo. Se gestiona el acceso a las estructuras de datos compartidas, y se verifica que el cliente que solicita la eliminación de dicho trabajo sea válido, y que tenga retenido el trabajo de la cola que se indica. De no ser de así, se retorna un error.

Es importante que esta operación se registre en el Logging garantizando de esta forma que ante una eventual caída, este trabajo no será reprocesado.

3) PasarToken (Id Cliente, Cola Fuente, Cola Destino, Trabajo) -> Ok?

Un trabajo se *pasa* de una cola "y" a otra "z", cuando se toma un trabajo para transformarlo, sin ser ésta la última etapa transformación. Luego de procesarlo, se pasa el trabajo de la cola "y" donde se encontraba retenido a una nueva cola "z" para así continuar con el esquema de transformaciones asociado. El pasaje de una cola a la siguiente debe ser una operación atómica para soportar la persistencia de los trabajos.

La función asociada al servicio de pasar un trabajo de una cola a otra es PasarToken. Recibe el identificador del cliente que tenía tomado el trabajo, el nombre de la cola donde se encuentra retenido el trabajo, el nombre de la cola a donde se va a agregar, y el identificador del trabajo. Esta función es similar a la composición de las dos funciones estudiadas previamente, BorrarToken y CrearToken, en una única. La razón por la cual se introduce una operación nueva en lugar de utilizar las operaciones de

BorrarToken y CrearToken, radica en la necesidad de que esta operación sea atómica. Si se ejecutara como dos sentencias separadas podría ocurrir una falla entre la ejecución de las mismas provocando la pérdida o inconsistencia del estado de los trabajos. Esta función se compone lógicamente de dos tareas primarias, la primera es la de eliminar el trabajo de la Cola Fuente, y la segunda es la de agregar el trabajo en la Cola Destino. La combinación de estas operaciones debe ser atómica, no pudiéndose eliminar un trabajo de la cola fuente sin agregarlo en la cola destino.

Se utiliza el recurso del Logging una vez que ambas operaciones fueron realizadas satisfactoriamente. Si falla en medio del pasaje de una cola a otra, lo último que se encuentra registrado es que el trabajo está en la cola fuente, ya que no se registró en el Logging el hecho de haberla borrado en el intento de pase fallido realizado.

Las operaciones BorrarToken y PasarToken que involucran un cliente que tenía retenido un trabajo, ejecutan rutinas de validación del cliente. Esta función pide acceso tanto a las estructuras de datos asociadas a la Cola Fuente como a la Cola Destino, ya que ambas van a sufrir modificaciones en sus datos. Asimismo, se ejecutan todas las rutinas de validación de los datos de entrada, se verifica la existencia el cliente y las colas, y que el trabajo se encuentre retenido en la Cola Fuente por el Cliente.

4) ParseArgs (*ID estructuras compartidas*)

La cuarta función de este módulo, se relaciona con la inicialización del entorno para la correcta ejecución de este servidor en sincronización con los demás componentes que conforman la herramienta.

El punto de entrada para el putserver es la función ParseArgs. Esta función se ejecuta cuando comienza este servidor y se encarga de parsear los argumentos que le pasa el connectserver al getserver en el momento de realizar el fork del mismo. El putserver obtiene los identificadores de las tablas en memoria compartida y del conjunto de semáforos que utilizará a través de estos parámetros. Esta función inicia el logging en el syslog. Una vez ejecutada la función ParseArgs, el putserver se transforma en un servidor que provee los servicios implementados por las funciones CrearToken, PasarToken y BorrarToken.

6.7 Implementación del servidor de logging: logserver

En la presente sección se describen detalles de implementación del servidor de logging y las funciones que lo componen, según el diseño de la sección 5.6 "Diseño del servidor de logging: Logserver", página 67.

El objetivo de este servidor es proveer un mecanismo seguro para almacenar en un medio persistente todas las operaciones significativas realizadas por los servidores que integran los managers. Es de fundamental importancia el cuidado de la performance de este servidor debido a que es un cuello de botella. Bloquea el accionar de todos los servidores hasta tanto no se haya efectuado la escritura en disco de las operaciones.

El servidor de logging actúa como servidor para todos los demás procesos que conforman los managers. Cada uno de los servidores es un cliente para el servicio que presta el logserver. El logserver no es un servidor de RPC, es un proceso que ejecuta como un demonio de Unix. Las funciones del logserver son invocadas por los servidores antes mencionados.

Se sincroniza con el historyserver mediante el uso de señales. El logserver le avisa al historyserver mediante una señal que se llenó un buffer y que puede procesar los buffers de log.

6.7.1.1 Detalle de las funciones del logserver.

Como se explicó en la sección 5.6.3.3 "Acciones realizadas por el logserver", página 75, la función única del logserver es la de almacenar en forma persistente las operaciones que le son enviadas por sus clientes. El logserver es un servidor, y como tal, tiene una etapa de inicialización y un ciclo de vida infinito recibiendo los pedidos de sus clientes y realizando el servicio solicitado. Efectúa las siguientes acciones:

1. Recibe como parámetros el identificador de la cola de mensajes creada por el connectserver que servirá para comunicar a todos los integrantes de los managers, el identificador de proceso (pid) del servidor de historia y los identificadores de los semáforos asociados a cada uno de los archivos de logging.
2. Abre los archivos de logging. Estos fueron previamente creados por el connectserver. De producirse cualquier error al abrir estos archivos, el servidor hace un *exit* provocando la muerte en cascada de los managers. De esta manera se asegura el no continuar procesando si no se pueden almacenar las operaciones realizadas. Ejecutar sin logging provocaría manejar información incompleta o incosistente.
3. Se ejecuta la función de entrada al servidor llamada *server()*. Esta es un ciclo infinito que recibe las solicitudes de servicio.

4. Cierra los archivos de logging. Esta acción no llega a ejecutarse cuando no se producen errores.

6.7.1.2 Implementación de la función server()

La única función que merece ser explicada en más detalle es la introducida en el punto 3 que realiza la función del servidor de logging.

int server()
Las acciones efectuadas son:

Se indica el tipo de mensaje a recibir por el logserver. Existe una única cola de mensajes a través de la cual se comunican todos los integrantes de los managers. Todos los mensajes que envían los managers (clientes) para ser procesados por el servidor de logging poseen el mismo tipo de mensaje. De esta manera el servidor extrae de la cola los mensajes destinados a él y no los consentimientos que él mismo envía a través de esta cola.

Se comienza a escribir por el primer archivo de logging disponible y se continúa escribiendo mientras haya lugar en el mismo. Este archivo es accedido por el logserver para registrar la operación y por el historyserver para luego procesar esa información. Antes de escribir, el logserver solicita exclusión mutua para el acceso a dicho archivo de logging, ya que el servidor de historia podría estar trabajando sobre el mismo. Si esto fuera así significa que no hay ningún archivo de logging disponible para ser usado y el servidor de logging se bloquea. Esto causa que todos los integrantes de los managers se bloqueen al realizar una operación, esperando el consentimiento del servidor de comunicación. Esta es una situación que no debería ocurrir si se consideraron apropiadamente la cantidad de archivos de logging. Si fuera necesario podrían agregarse más.

Como primera acción antes de almacenar operaciones en el archivo de logging, se escribe en el mismo un timestamp, indicando la fecha y hora de proceso. Esto servirá como verificación del orden en que deba ser recuperado por el servidor de historia.

Una vez finalizadas las acciones de inicialización se comienzan a recibir pedidos. Cada mensaje que se recibe, se procesa. Para ello se utiliza la función *proceso_mensaje* que escribe el mensaje en disco en forma sincrónica. Retorna el tipo de cliente que lo envió para poder mandarle el consentimiento a ese cliente. De resultar satisfactoria la escritura, se arma y envía el consentimiento. De existir algún error al recibir el mensaje o al enviar el consentimiento, se hace un *exit*, provocando la muerte en cascada de los managers, por las razones previamente explicadas.

Una vez que se completa un archivo de logging, se libera el lockeo del mismo para que pueda ser procesado por el servidor de historia. Se cambia al próximo archivo de logging y se le avisa al servidor de historia que tiene trabajo. Esto último se realiza enviando una señal al identificador de proceso del historyserver con la señal SIGUSR1. Se utiliza la llamada al sitema: kill(hispid, SIGUSR1).

6.7.1.3 Acceso de los clientes al logserver

Como se explicó en la sección 5.6.3.4 "Acceso de los clientes al logserver", página 75, el servidor de logging actúa como servidor para todos los demás procesos que conforman los managers. La función usada por los clientes del logserver para acceder a sus servicios se denomina *writelog* y es la encargada de enviar un mensaje al logserver conteniendo la operación a ser almacenada en el archivo de logging para persistencia. El cliente que llama a esta función se bloquea hasta no recibir el consentimiento del servidor. Cuando *writelog* retorna, ya se ha salvado en almacenamiento persistente la operación.

Esta función es sumamente sencilla:

```
int writelog (int cliente, Qtype cola, char * trabajo)
```

1. Se setea el tipo de mensaje indicando que debe ser recibido por el servidor de logging, se indica el cliente que envía el mensaje, la cola sobre la que se realizó la operación y el nombre del trabajo involucrado.
2. Se rellena el mensaje con blancos para que todos los mensajes sean de la misma longitud. De esta manera, el logserver simplemente recibe el mensaje y lo escribe en disco, sin necesidad de procesarlo previamente. Es importante tener en cuenta que las operaciones sobre cadenas de caracteres consumen tiempo y así el servidor de logging se libera de esta tarea.
3. Se envía el mensaje.
4. Se espera hasta recibir el consentimiento del servidor de logging.

En caso de existir algún error en el envío o recepción de los mensajes indicados en 3 y 4 respectivamente se realiza un *exit* del servidor invocante de esta llamada, produciéndose la muerte en cascada de los managers. No es posible continuar procesando en caso de un error de este tipo ya que se perderían trabajos, produciéndose inconsistencia en los trabajos.

Las funciones para enviar y recibir mensajes utilizadas fueron escritas para no depender del sistema operativo en el cual se haga la llamada a la cola de mensajes. Se utilizan funciones que encapsulan los detalles de implementación de los mecanismos de IPC de UNIX. De cambiar de sistema operativo, sólo deberían cambiarse estas funciones de alto nivel.

6.8 Implementación del servidor de historia: historyserver

En la presente sección se describen detalles de implementación del servidor de historia y las funciones que lo componen como se diseñó en la sección 5.7 "Diseño del servidor de historia: Historyserver", página 76.

El objetivo de este servidor es soportar y garantizar la recuperación ante fallas del sistema trabajando en cooperación con el logserver.

Este módulo es el responsable de aplicar la semántica asociada a un movimiento de trabajos, a los archivos que deja el servidor de logging. Si un trabajo ha sido pasado de la Cola X a la Cola Y, hay que eliminar todas las ocurrencias que ubican al trabajo en la Cola X y simplemente almacenar la información que indica que el trabajo está en la Cola Y.

El historyserver no presta servicios a los clientes de la herramienta y ejecuta como un demonio de Unix. Se sincroniza únicamente con el logserver mediante el uso de señales. Este servidor ejecuta cuando el logserver le avisa mediante una señal que se llenó un buffer y que, por lo tanto, el historyserver tiene trabajo que realizar. El historyserver no exporta funciones para que sean invocadas por otros procesos, sino que realiza sus tareas dentro del manejador de señales.

El historyserver comienza su trabajo cuando el logserver le avisa que ha llenado un buffer y que el mismo está disponible para ser procesado. El procesamiento consiste en leer cada par <trabajo,cola> contenido en el archivo de historia y aplicar sobre el nombre de cada trabajo una función que calcule una clave de hashing. Esta clave es la usada para acceder a la tabla de hashing que almacena los pares <trabajo, cola>. Como sólo es necesario almacenar la última posición de un trabajo en una cola, a medida que se procesa el archivo de historia, se modifica la dupla de la tabla de hashing para contener la última cola que contiene al trabajo. Al finalizar el procesamiento del servidor, los pares mencionados en la tabla de hashing representan la imagen del sistema.

Para mantener la consistencia de la información es fundamental leer el buffer en el orden de producidos los movimientos de trabajos. Por la forma en que graba el logserver, se debe leer de arriba hacia abajo y siempre luego de haber leído la información histórica almacenada en el archivo de historia.

Una vez procesadas todas las entradas del buffer, la tabla de hashing refleja la nueva imagen del sistema y debe ser salvada en un archivo persistente para ser utilizada en caso de caídas. Es necesario que una vez que el nuevo estado se encuentra correctamente salvado en un archivo persistente se pueda, en caso de falla, recuperar el estado de ejecución de la herramienta a partir del mismo.

6.8.1.1 Detalle de las funciones introducidas

Este módulo consta de cuatro funciones principales.

1) Main( )

El Main() es la función encargada de manejar la sincronización con el servidor de logging, y de iterar en espera de buffers a procesar.

Para establecer la sincronización entre el servidor de historia y el de logging, se utiliza el mecanismo de señales que provee Unix. El historyserver se bloquea en espera de una señal generada por el logserver para indicarle que se completó un nuevo buffer y que debe ser procesado.

Inicializa las variables asociadas a la cantidad y descriptores de los buffers que maneja el sistema. Luego setea la señal para poder sincronizarse con el servidor de logging y disparar el procesamiento de los buffers. Una vez realizadas estas tareas entra en un loop infinito bloqueándose hasta recibir un signal, y al recibirlo desencadena las siguientes acciones:

1. Realiza un P sobre el semáforo asociado al buffer para obtener exclusión mutua sobre el mismo.
2. Llama a CompressHistory con la Tabla de Hashing (esta tabla está vacía) y el archivo de historia actual para procesar la historia que se tenía hasta el momento.
3. Llama a CompressHistory con la Tabla de Hashing (en este momento contiene la información obtenida del archivo de historia) y con el buffer.
4. Llama a FlushHistory con la Tabla de Hashing ya consolidada y el archivo de historia donde se debe salvar el nuevo estado.
5. Llama a PonerEnBlanco para limpiar el buffer ya procesado.
6. Libera el derecho de exclusión mutua sobre el buffer ejecutando un V sobre el semáforo asociado.
7. Se bloquea en espera de un otra señal.

2) CompressHistory(Tabla de Hashing, Archivo de Historia o Buffer) -> Tabla de Hashing

La función CompressHistory lee las duplas <trabajo, cola> del archivo de historia y de los buffers y arma consistentemente la tabla de hashing, reflejando los movimientos que se fueron produciendo en el sistema. La función recibe como parámetros la tabla de hashing y el archivo de historia o buffer que le corresponde procesar. Existe más de un buffer y previo a cada buffer hay que procesar el archivo de historia existente hasta ese momento.

Abre el archivo o buffer para lectura, y lee las duplas que lo componen. Tanto el archivo de historia como los buffers tienen el mismo formato, permitiendo generalizar esta función para el procesamiento de ambos. Por cada dupla que se lee, se extrae el nombre del trabajo y en base a él se calcula la clave de hashing utilizada para acceder a la tabla. Una vez que se tiene la posición se comienza a recorrer la lista de trabajos asociados en busca de otra ocurrencia del mismo. En caso de no encontrarse inserta esta nueva upla. Caso contrario, la upla que fue leída es la que contiene la información más reciente, por lo tanto se debe actualizar la upla que se encontraba antes en la Tabla de Hashing con el nuevo nombre de cola leído.

Cuando llega al final del archivo, da por concluída su tarea y retorna el control con la Tabla de Hashing que acaba de armar.

3) FlushHistory(Tabla de Hashing, Archivo de Historia) -> Ok?

Como complemento a CompressHistory, una vez que se termina de procesar el buffer y se completa la tabla de hashing, la información se almacena en un archivo persistente para mantenerse disponible en caso de fallas. La función que realiza esta tarea es FlushHistory y recibe como parámetros la tabla de hashing y el archivo persistente donde se deberá salvar la información.

Se encarga de salvar toda la información que se encuentra en la Tabla de Hashing, que fue correctamente consolidada y que refleja el estado actual del sistema a un medio persistente para que pueda ser utilizado como fuente para reiniciar el sistema en caso de caídas.

Podría suceder que en el momento en que se está sobrescribiendo el archivo de historia con el nuevo estado se produzca una caída y así perder toda la historia, la nueva y la vieja. Para evitar esto, antes de abrir el archivo para escritura, acción que truncaría el archivo, se ejecuta una función que copia estos datos en un archivo de resguardo o backup. De esta forma, con la historia vieja, y con el buffer conteniendo los últimos movimientos se puede reconstruir nuevamente el estado.

Para finalizar recorre una a una las entradas de la tabla, salvando estos datos en el archivo de historia. En el momento de guardar se verifica que la cola no tenga el identificador de Borrado, en cuyo caso se saltea dicha upla, evitando registrarla en la imagen final del estado del sistema.

4) Poner en Blanco (Buffer) -> Ok?

Una vez que el nuevo estado fue satisfactoriamente salvado y no hay peligro de perder información, se procede a blanquear el buffer de logging recién procesado de manera tal que el servidor de logging pueda nuevamente registrar transacciones en el mismo. La función que se encarga de inicializar nuevamente el buffer es PonerEnBlanco y obtiene como parámetro el identificador del buffers a blanquear. Esto se realiza para dejar el buffer en condiciones de ser utilizado por el servidor de logging para continuar con su trabajo. Sólo se ejecuta cuando se está seguro que el estado fue salvado satisfactoriamente. Se considera inicializar un buffer al hecho de ponerlo en blanco.

6.9 Implementación del servidor de administración: admserver

En la presente sección se describen detalles de implementación del servidor getserver y las funciones que lo componen, según lo diseñado en la sección 5.8 "Diseño del servidor de administración: Admserver", página 80.

El admserver es un servidor de RPC. Este servidor provee dos clases de servicios: el primero está orientado a los clientes de la herramienta y el segundo está destinado a los administradores y operadores del sistema administrado por la herramienta.

El admserver es el servidor de los managers al que los clientes se conectan primero. Este servidor, mediante la función Present, permite a los clientes registrarse para el uso futuro de los managers.

Este servidor provee funciones para monitorear y administrar la herramienta y los objetos que la misma maneja. Como las estructuras donde se almacenan el estado de los procesos, las colas y los trabajos son tablas en memoria con un sistema restringido de acceso, se diseñó un servidor interno a la herramienta para proveer las funciones de monitoreo y administración.

6.9.1.1 Detalle de las funciones del admserver.

Este módulo consta de ocho funciones de monitoreo y administración, una que permite que nuevos clientes se registren y una de inicialización llamada ParserArgsAdm.

Conexión de un cliente a la herramienta.

1) Función Present

La función Present recibe el pedido de un cliente nuevo que desea hacer uso de la herramienta y lo da de alta en el sistema. Para incorporarlo al sistema se requiere conocer el host o máquina donde ejecutará el cliente y el tipo de proceso que este cliente utilizará. El tipo de proceso determina a que colas tendrá acceso el cliente.

El ingreso en las tablas del sistema se realiza mediante la función DarAltaClient().

En el mecanismo a través del cual el cliente se presenta a los managers participan varios módulos y servidores de la herramienta. El cliente se comunica localmente con un módulo de los managers que ejecuta el la máquina del cliente, este módulo llamado watchdog, se comunica con el monitor quien solicita al admserver el alta del nuevo cliente. Este mecanismo se estudió en la sección 5.10 "Esquema de la conexión de un cliente de la herramienta con los managers".

Borrado de un cliente.

2) Función ClientDown

La función ClientDown es invocada cuando un cliente de la herramienta muere. El mecanismo implementado por monitor/watchdog permite conocer esta situación. La función ClientDown es la encargada de actualizar las tablas de los managers para liberar todos los recursos del cliente. Se necesita conocer el host donde estaba ejecutando el cliente y el identificardor del mismo dentro del sistema. Con estos datos debe:

1. Liberar los trabajos que el cliente tenía tomados para ser procesados
2. Eliminar el cliente de la tabla de clientes

Es importante cambiar el identificador de la entrada en la tabla lo antes posible para evitar inconsistencias.

Información ordenada por tipo de proceso.

3) Función DumpTablesCl

La función DumptablesCl pertenece a la categoría de servicio de administración. Mediante esta función es posible acceder a las tablas de memoria y obtener la información de todos los clientes conectados con un determinado tipo de proceso, los trabajos que tienen tomados y la cola a la que pertenece cada trabajo.

A través de la función MapearProcType() se averigua el idenificador de tipo de proceso correspondiente al nombre de tipo de proceso. Se recorre la tabla de clientes buscando aquellos cuyo identificador de tipo de proceso concuerde con el buscado. Con estos datos se accede a la tabla de clientes y se busca la información de los trabajos tomados.

Información de colas.

4) Función DumpTablesQ

La función DumpTablesQ pertenece a la categoría de servicio de administración. Mediante esta función es posible acceder a las tablas de memoria y conocer las colas existentes en el esquema. Se recorre la tabla de colas y para cada una de ellas se registra el nombre y el identificador.

Información de tipos de proceso.

5) Función DumpTablesProc

La función DumpTablesProc pertenece a la categoría de servicio de administración. Mediante esta función es posible acceder a las tablas de memoria y conocer los tipos de proceso existentes en el esquema. Se recorre la tabla de procesos y para cada una de ellos se registra el nombre.

Existencia de un trabajo determinado en una cola.

6) Función ExisteToken

La función ExisteToken pertenece a la categoría de servicio de administración. Mediante esta función es posible conocer si un determinado trabajo existe en una cola especificada. Primero se averigua el identificador de la cola especificada, si la misma existe. A través de la función SeekToken() a la que se le pasan como parámetros el nombre del trabajo y el identificador de la cola, se averigua si el trabajo está en la cola.

Búsqueda de un trabajo.

7) Función BuscarTokenEnQueues

La función BuscarTokenEnQueues pertenece a la categoría de servicio de administración. Mediante esta función es posible conocer si un determinado trabajo existe en el esquema y averiguar en qué cola se encuentra. A diferencia de la función ExisteToken, busca en todas las colas definidas en el esquema. Para cada cola se ejecuta la función SeekToken().

Búsqueda de un conjunto de trabajos.

8) Función VerLote

La función VerLote pertenece a la categoría de servicio de administración. Mediante esta función es posible obtener una lista de todos los trabajos de una cola cuyos nombres cumplen una determinada condición. Se le pasa como parámetros el patrón que debe cumplir el nombre del trabajo y el nombre de la cola. Una vez averiguado el identificador de cola correspondiente al nombre de cola especificado se llama a la función ArmarVerLote. Esta función recorre la cola y retorna el nombre de cada trabajo que concuerde con el patrón especificado.

Cantidad de trabajos por cola.

9) Función ContarJobs

La función ContarJobs pertenece a la categoría de servicio de administración. Mediante esta función es posible conocer la cantidad de trabajos encolados en una determinada cola.

6.10 Implementación del módulo monitor.

El monitor, en colaboración con el watchdog, se encarga de monitorear el funcionamiento de los clientes de la herramienta. El monitor trabaja en combinación con los watchdogs que ejecutan en cada máquina cliente para actualizar el estado de los clientes ante falla o muerte de los mismos. Verifica a cierto intervalo de tiempo que todos los clientes que tiene registrados siguen operando. En caso de muerte de algún cliente libera los recursos tomados por el mismo y devuelve los trabajos en proceso a las colas de donde fueron tomados.

El monitor participa en el mecanismo de presentar un cliente a los managers.

El monitor no es un servidor de RPC y no presta servicios en forma directa a los clientes de la herramienta. No posee ninguna función que sea invocada directamente por los clientes.

El módulo monitor está integrado por tres procesos: el proceso *monitor* es el padre de los otros dos procesos llamados *assign* y *ping*. Los tres procesos ejecutan en cooperación, como demonios de Unix y corren independientemente de los servidores que son servidores de RPC. Se comunica con los watchdogs a través del protocolo TCP/IP e invoca funciones del admserver vía RPC.

6.10.1.1 Implementación de las funciones del monitor.

Organizaremos esta sección describiendo cada uno de los tres procesos que integran el módulo monitor y su interacción.

Todos los procesos que integran el monitor loguean sus acciones y estados permitiendo el debbugging del accionar de los mismos. Esto es importante por el esquema de procesos clientes distribuidos en distintas máquinas que se desea monitorear. Se soportan diferentes niveles de debugging que producen distinto nivel de detalle en la información loggeada.

6.10.1.1.1 Monitor

El proceso monitor se encarga de crear los otros dos procesos que conforman el módulo monitor y liberar los recursos de los mismos al terminar la ejecución ya sea en forma normal o anormal. Los procesos assign y ping son los que realizan el trabajo específico de monitorear a los clientes.

Las funciones llevadas a cabo por el monitor se basan en llamadas al sistema Unix que permiten la creación de los procesos ping y assign y el seteo de la comunicación entre

los mismos a través de mecanismos de IPC (Inter Process Communication) y funciones del protocolo TCP/IP.

La siguiente es una lista de las funciones llevadas a cabo por el monitor. No constituye una secuencia de acciones o ejecución. La misma ya fue presentada en la sección de diseño de este proceso.

1. Abre el archivo de log donde se almacenan las acciones y estados relevantes del monitor. El nivel de detalle de esta información es parametrizable.
2. Establece el manejador de señales compartido por los tres procesos para la señal SIGHUP. El manejador de señales resetea el monitor. Resetear el monitor significa borrar toda la información de clientes y máquinas contenida en las tablas del monitor.
3. El monitor se presenta a los managers para convertirse en un cliente de RPC del admserver.
4. Crea las tablas de memoria compartida utilizadas por los tres procesos y las inicializa.
5. Crea el semáforo para sincronizar el acceso de los tres procesos a las tablas.
6. Realiza el fork del proceso que ejecuta la función assign.
7. Realiza el fork del proceso que ejecuta la función ping.
8. Inicializa el manejador de señales para las señales SIGTERM o SIGINT. De recibirlas interrumpe y termina la ejecución de los tres procesos que conforman el módulo monitor.
9. El monitor contiene el código de dos funciones que son invocadas por los procesos assign y ping. Una de ellas libera la información de un determinado cliente y la otra libera la información de todos los clientes ejecutando en una máquina determinada. Estas funciones se ejecutan al reconocerse la muerte de un cliente.

6.10.1.1.2 Assign.

El proceso assign es el proceso del módulo monitor que participa en el mecanismo de presentación de un nuevo cliente a los managers. El assign escucha en el port en el que el watchdog escribe para solictar un Present de un nuevo cliente. Con la información que envía el watchdog local al cliente, el assign gestiona el alta del cliente a través del admserver.

Las acciones realizadas son:

1. Realizar el attach de las tablas de memoria compartida.
2. Crear el socket para recibir.
3. Recibe pedidos del watchdog para presentar un nuevo cliente y los procesa, en un ciclo mientras vivan los managers.

Estudiaremos a continuación la manera de procesar los pedidos para presentar un cliente. Antes de ejecutar la función para procesar el Present, el assign ejecuta un P del semáforo que protege las tablas compartidas. Al finalizar la ejecución de la función lo libera. El assign efectúa las siguientes acciones para procesar el Present:

1. Recibe los parámetros envíados por el watchdog. Los mismos son el identificador de proceso del watchdog local a la máquina donde ejecuta el postulante a cliente, el tipo de proceso que utilizará el cliente que quiere conectarse a los managers y el nombre de la máquina donde ejecuta el cliente.
2. Busca el nombre de la máquina enviado dentro de las tablas del monitor.
   a) Si no tiene información sobre esta máquina la agrega a la tabla de hosts especificando el identificador de proceso enviado como parámetro como identificador del watchdog local a esa máquina.
   b) Si tiene información de esa máquina verifica que el identificador de proceso del watchdog enviado como parámetro coincida con el almacenado en las tablas del monitor. Si no coincidiera significa que el watchdog que ejecutaba en esa máquina murió y una nueva instancia del watchdog está en ejecución. La máquina pudo haber sufrido una falla o caída y la información almacenada en las tablas internas puede estar desactualizada. Aunque los clientes no hubieran muerto deben presentarse al nuevo manager. Por lo tanto, es necesario borrar la información de todos los clientes ejecutando en esa máquina y actualizar la información de la tabla de hosts con el identificador de proceso del nuevo watchdog. Se debe limpiar la tabla de clientes que mantiene el monitor pero también la información de los clientes que mantienen los managers en sus propias tablas. Para ello se invoca a la función ClientDown del admserver. Referirise a las secciones de Diseño e Implementación del Admserver para detalles del funcionamiento de dicha función. Al eliminarse la información de las tablas de los managers los identificadores de cliente se tornan inválidos. Si hubiera algún cliente que sobrevivió no podrá seguir operando con los managers y deberá presentarse nuevamente. Esta medida es precautoria para evitar que clientes viejos de otra era de los managers pudieran causar inconsistencias: si un cliente viejo tenía trabajos en su poder y se produjo una caída o terminación normal de los managers, esos trabajos son devueltos a las colas de donde los tomó el cliente. Si luego de reiniciarse los managers ese cliente viejo deseara realizar alguna operación con ese trabajo recibirá un error porque su identificador ya no es válido, debiéndose presentar a los nuevos managers para continuar operando.
3. El assign invoca la función PresentToManager del admserver pasando como parámetro el tipo de proceso y el admserver le retorna un identificador válido para el nuevo cliente.
4. Si el admserver retorna un identificador, el assign agrega el cliente a las tablas del monitor.
5. Envía el identificador del nuevo cliente de la herramienta al watchdog ejecutando en la máquina del cliente. Si no puede enviar el mensaje significa que el watchdog falló y se eliminan todos los clientes de esa máquina de las tablas internas del

monitor y se envía un ClientDown al admserver por cada cliente ejecutando en esa máquina para actualizar las tablas de los managers.

6.10.1.1.3 Ping.

El proceso ping es el proceso del monitor encargado de monitorear el funcionamiento de los clientes de los managers. El ping envía un mensaje a un intervalo de tiempo parametrizable a los watchdogs de las máquinas donde ejecutan clientes de los managers. Si encuentra un error en los watchdogs o en los clientes se encarga de eliminar las entradas de las tablas para los mismos.

Este proceso posee una función principal encargada de realizar esta tarea y otras funciones que colaboran para realizarla. Las acciones realizadas son:

1. Realiza el attach de las tablas en memoria compartida.
2. En un ciclo infinito mientras vivan los managers, duerme durante el intervalo especificado y luego envía un mensaje a cada una de las máquinas contenidas en las tablas del monitor y que ejecutan clientes de los managers.

Estudiaremos la manera de monitorear el funcionamiento de los clientes de los managers. Se ejecuta una función denominada ping para cada una de las máquinas registradas en las tablas internas como ejecutando clientes de los managers. Se llama ping porque tiene un objetivo similar al comando o función ping: saber si responde. En este caso se desea saber que la máquina responde y además solicitar al watchdog ejecutando en dicha máquina información referida a los clientes de la herramienta.

Hay que tener especial cuidado con la utilización de exclusión mutua sobre las tablas del monitor durante la ejecución de esta función. La comunicación con la máquina a monitorear puede tardar y no sería lógico bloquear el acceso a las tablas compartidas durante este proceso porque se estaría bloqueando todo el módulo monitor.

El proceso ping para procesar la función ping efectúa las siguientes acciones:

1. Realiza un P del semáforo que protege las tablas compartidas y copia la información contenida en el registro de la máquina a monitorear. Almacena el identificador de proceso del watchdog y el nombre de esa máquina.
2. Para verificar el buen funcionamiento del watchdog de la máquina, se abre el socket, se envía un mensaje al socket del watchdog con un código que le indica al mismo que se trata de una verificación del monitor y se espera la respuesta.
3. El monitor espera respuesta del watchdog. Hay varias posibilidades:
    a) Se recibe respuesta válida:

i. El watchdog envia un mensaje conteniendo los identificadores de los clientes de la máquina donde él ejecuta que han caído o un código especial si no falló ningun cliente.
ii. Se parsea la respuesta y para cada cliente que murió se elimina de las tablas del monitor y se invoca a la función ClientDown del admserver para que lo elimine de las tablas compartidas por los managers.

b) No se recibe respuesta o es inválida o se recibe respuesta pero cambió el identificador de proceso del watchdog que envía la respuesta:

i. Se ejecuta un P del semáforo que proteje las tablas compartidas.
ii. Si la tabla no contiene los datos que el ping copió de la misma (ver punto 1) significa que ya fue actualizada y no deben eliminarse los clientes. Si la tabla todavía contiene los datos que el ping copio de la misma se eliminan todos los clientes de esa máquina y para cada uno de ellos se invoca un ClientDown al admserver para que lo elimine de las tablas compartidas por los managers.
iii. Si no hubo respuesta o fue inválida se elimina la máquina de la tabla de máquinas de monitor porque se considera que la misma no responde. Si hubo respuesta se actualiza el identificador de proceso del watchdog en la tabla de máquinas.
iv. Se libera el semáforo que proteje las tablas compartidas.

6.11 Implementación del módulo watchdog.

El watchdog, en colaboración con el monitor, se encarga de monitorear el funcionamiento de los clientes de la herramienta. En caso de muerte de alguno de ellos es responsable de avisar al monitor.

Es un proceso local a cada una de las máquinas donde ejecutan los clientes. Es el módulo de los managers que ejecuta distribuido con respecto a los servidores que integran la herramienta. Hay tantos watchdogs como máquinas donde se ejecuten clientes de la herramienta y un único watchdog por máquina. Los watchdogs ejecutan en forma distribuida para poder monitorear, en combinación con el monitor, los clientes y actualizar el estado de los mismos ante falla o muerte.

Si el watchdog no respondiera al monitor, la mera ausencia de respuesta indica al monitor que la red de comunicación falló, la máquina remota donde ejecuta el watchdog falló o el watchdog ha caído. Esta información permite al monitor desencadenar la elimininación de las tablas de los managers de todos los clientes que ejecutaban en esa máquina.

El watchdog participa en el mecanismo de presentar un cliente a los managers. El watchdog recibe el pedido de Present() de un nuevo cliente, transmite el pedido al monitor y recibe el identificador que envía el monitor otorgado por el admserver y se lo comunica al nuevo cliente.

El watchdog no presta servicios en forma directa a los clientes de la herramienta y no posee ninguna función que sea invocada por los mismos.

El watchdog es un único proceso que ejecuta en diferentes entornos y sistemas operativos. El mismo código fuente de watchdog puede recompilarse para ser ejecutado en diferentes sistemas operativos. En el presente trabajo de tesis se soportan watchdogs para ser ejecutados en varias versiones de Unix como ser AIX y Linux, así como OS/2 y DOS.

El watchdog ejecuta distribuido de los managers e independientemente de los servidores que son servidores de RPC. Ejecuta como un proceso demonio y se comunica con el monitor y los clientes a través del protocolo TCP/IP.

6.11.1.1 Implementación de las funciones del monitor.

La siguiente es una lista de las funciones llevadas a cabo por el watchdog. No constituye una secuencia de acciones o ejecución.

1. Si el entorno es Unix se setea el usuario efectivo de ejecución como root.
   Se inicializa el proceso como un demonio que ejecuta una única instancia en la máquina. Esta acción es realizada por una función de biblioteca que se halla en

libdaemon.a. Realiza las acciones necesarias para convertir al proceso que la ejecuta en un demonio. La función es **InicializarProceso(ArchivoDeLock).**
2. Establece el manejador de las señales de SIGTERM y SIGINT para interrumpir la ejecución del watchdog.
3. Si el entorno es Unix establece el manejador de la señal SIGHUP para resetear el watchdog en caso de ser recibida. Resetear el watchdog significa eliminar todos los datos de clientes contenidos en la tabla del watchdog.
4. Averigua el nombre del host local donde está ejecutando.
5. Crea la conexión TCP y escucha sobre la misma.
6. En un ciclo infinito recibe mensajes en el socket que creó y realiza las acciones determinadas por el tipo de mensaje recibido. Las acciones solicitadas al watchdog son registrar un nuevo cliente, verificar los clientes que ejecutan en la máquina, informar el nombre real de la máquina o informar el nombre del servidor donde ejecutan los managers.

En esta sección se detallarán cada una de las funciones que cumple el watchdog y que se mencionaron en el punto 6.

Registrar un nuevo cliente.

El watchdog participa en el mecanismo de presentar un cliente a los managers. Para una visión más global del mecanismo referirse a la sección 5.10 "Esquema de la conexión de un cliente de la herramienta con los managers".

El watchdog participa en este mecanismo a través de las siguientes acciones:

1. Recibe el mensaje enviado por el cliente. Los parámetros enviados en el mismo son el identificador de proceso del cliente y el tipo de proceso como el cual el cliente quiere conectarse a los managers.
2. Busca lugar en la tabla de clientes del watchdog. Si no hubiera lugar envia un mensaje de error al cliente especificando error del watchdog.
3. Pide un identificador de la herramienta para este cliente: envia un mensaje al monitor avisando que un nuevo cliente se presentó a los managers. Para ello, crea una conexión TCP para comunicarse con el monitor. Envia un mensaje conteniendo el identificador de proceso del watchdog, el tipo de proceso solicitado por el cliente y el nombre de la máquina donde está ejecutando.
4. El watchdog espera hasta recibir un mesnaje del monitor con el identificador del cliente. La respuesta del monitor contiene un código de mensaje y el identificador del cliente otorgado por el admserver para la herramienta.
5. Si se recibió un identificador de cliente inválido o hubo algún error se le envia un mensaje de error al cliente que solicitó el present. Si se recibió un identificador válido se agregan en la tabla del watchdog el identificador otorgado por los managers y el identificador de proceso (pid) del nuevo cliente. Se envia al cliente mediante un mensaje de TCP el identificador de la herramienta otorgado y el nombre de la máquina donde ejecutan los managers.

Verificar los clientes que ejecutan en la máquina.

El monitor verifica a cierto intervalo de tiempo que todos los clientes que tiene registrados sigan operando. Para ello utiliza los servicios de los watchdogs locales a las máquinas donde ejecutan los clientes a verificar. Para conocer el estado de los clientes, el monitor envía un mensaje al watchdog quien verifica el estado de los clientes y avisa al monitor en caso que alguno de ellos haya caído.

Las acciones mediante las cuales el watchdog verifica los clientes a pedido del monitor son las siguientes:

1. Para cada uno de los clientes registrados en la tabla del watchdog se verifica si el mismo sigue ejecutando.
2. Se verifica el proceso cliente. Si el mismo murió se libera su entrada en la tabla del watchdog. Para verificar si el proceso sigue ejecutando en un entorno Unix se envia una señal inofensiva mediante la llamada al sistema *kill* al proceso con el identificador de proceso del cliente a verificar. El código de retorno de la llamada informa si el proceso existe. Si el entorno es OS/2, al no existir una llamada al sistema análoga al kill, se debió encontrar otra manera menos ortodoxa de verificar la existencia del proceso. Se optó por tratar de cambiar la prioridad al proceso. De acuerdo al código de retorno de la llamada se sabe si el proceso existe.
3. Se construye un mensaje con el identificador de proceso del watchdog y los identificadores de la herramienta de los clientes que murieron. Si todos siguen ejecutando el mensaje contiene el identificador de proceso del watchdog y un código especial que indica que todos los clientes siguen ejecutando.

Informar el nombre real de la máquina.

Esta función retorna el nombre del host donde está ejecutando el watchdog. Esta información es necesaria para inicializar la tabla de máquinas del monitor y poder tener un dato adicional al identificador del proceso del watchdog como identificador de una máquina donde ejecutan clientes de la herramienta. En caso de muerte y reinicialización del watchdog, el monitor puede conocer la máquina por nombre sin depender de la identidad del watchdog que puede variar.

Informar el nombre del servidor donde ejecutan los managers.

Esta función retorna el nombre de la máquina donde se ejecutan los managers. Esta información es importante para conocer la localización de los servidores RPC que la integran.

7. Manual del usuario.

Como desprendimiento de la implementación se escribió un manual para el usuario, el mismo consta de dos partes, destinadas a diferentes tipos de lectores. La primera es para los desarrolladores que puedan hacer uso de las APIs para uso de la herramienta y la segunda está destinada al administrador de la misma.

7.1 Primera parte

APIs para los clientes

7.2 Conceptos necesarios para utilizar las funciones provistas:

El sistema consta de colas de trabajos que son manejadas por los managers. Estos actúan como distribuidores de trabajos, garantizando no entregar el mismo trabajo a más de un cliente y evitando la pérdida de trabajos ante posibles caídas. Desde el punto de vista de los cliente que usarán estas funciones para obtener trabajos a procesar hay algunos conceptos claves:

Tipo de proceso: Cada cliente que desee conectarse con el manager debe hacerlo como algún "tipo de proceso" predefinido. El tipo de proceso especifica a los managers los permisos y accesos de los diferentes procesos a los recursos del sistema, y por lo tanto a las diferentes colas y trabajos. Estos tipos de proceso serán especificados por el administrador del sistema. El administrador le informará al programador que "tipo de proceso" debe usar, dependiendo de las colas que usará de entrada y de salida. Si un proceso desea conectarse con una cola no permitida para su correspondiente tipo de proceso, se retornará error.

Cola: Las colas son los repositorios temporales de los trabajos. A cada tipo de cola se asocia un trabajo en un estadio distinto de procesamiento o un tipo de dato o información. Cada proceso que desee transformar un trabajo lo toma de alguna de sus colas de entrada, lo modifica y lo coloca en alguna de sus colas de salida. Dependiendo de la arquitectura de la aplicación a cada cola se asocian el mismo o diferentes tipos de datos.

Tipos de colas:

Entrada: Son las colas desde las cuales los clientes toman trabajos para procesarlos.

Salida: Son las colas donde los clientes depositan sus trabajos una vez procesado el dato de entrada y generado el dato de salida.

Inicial: Son las colas de entrada al sistema. En estas colas se crean trabajos que antes no existían para el sistema. Es decir, se introducen nuevos trabajos en el flujo de datos a procesar.

Cuando un cliente de los managers comienza su ejecución debe seguir algunos pasos para que los managers lo reconozcan como cliente y para poder utilizar las colas para manipular trabajos:

1. Presentarse a los managers: mediante este paso se crea un nuevo identificador de cliente para los managers. Desde este momento este proceso es un cliente y es monitoreado como tal avisando de posibles caídas, liberando sus trabajos pendientes en caso de muerte, etc.
2. Conectarse con cada una de las colas con las que desee interactuar: Se debe conectar a todas las colas de entrada y/o salida. Los managers ya conocen cuales están permitidas y cuales actúan como entrada y/o salida ya que el proceso se presentó indicando su tipo de proceso.
3. Puede utilizar las funciones de manipulación de trabajos: CrearToken, TomarToken, PasarToken, BorrarToken explicadas más abajo.

7.3 Funciones Provistas

A continuación se presentan las funciones que permiten a los clientes hacer uso de las operaciones soportadas por la herramienta. Estas funciones constituyen las APIs que la herramienta provee a los clientes.

```
rc = Present(char * tipoproc, IdType * id);
```

Antes de comenzar la ejecución del proceso cliente, se ejecuta esta función para que el servidor realice un alta en su tabla de procesos clientes. Se especifica el tipo de proceso que utilizará el cliente. Este determina las colas con las que se puede conectar el cliente y operaciones puede realizar sobre las mismas.

Como parámetro de salida se recibe un identificador único, que el proceso tendrá que utilizar cada vez que quiera comunicarse con los managers.

```
tipoproc:   Nombre del tipo de proceso que se desea conectar.
            Parámetro de Entrada.
id      :   Handle del cliente. Parámetro de Salida.
rc      :   Entero que contiene el resultado de la
            operación. Código de retorno de la función.
```

```
colaPtr = ConectarCola(IdType id, char * nomcola, char
  *dir);
```

Esta función inicializa los recursos y hace accesible la cola al cliente. El primer parámetro es el identificador de cliente (id) devuelto por la función Present. El segundo parámetro es un string que especifica el nombre de la cola con la que se quiere conectar. El tercer parámetro retorna el nombre del directorio (path completo) asociado a la cola. De este directorio se leerán los datos si es usada como cola de entrada o se escribirán si es usada como cola de salida.

Retorna un puntero o handle a la cola con la que se desea conectar. Se utilizará este identificador de la cola en futuras llamadas al manager. Se utiliza un identificador para que el servidor, en subsecuentes requerimientos pueda acceder directamente a la cola y no deba realizar una búsqueda por el nombre de la cola. Si hubo error retorna NULL.

```
id      : IdType que identifica al cliente. Parámetro de
          Entrada.
nomcola : Puntero a carácter donde se encuentra el nombre de
          la cola con la que se desea conectar. Parámetro de
          Entrada.
dir     : Cadena de caracteres que contiene el directorio
          (path completo) asociado a la cola. Esto contiene
          el path completo sin la barra del final. De allí
          se levantarán los datos si es usada como cola de
          entrada o se escribirán si es usada como de
          salida. Parámetro de salida.
colaPtr: Puntero o handle de la cola. Es de tipo Cola.
```

`rc = CrearToken(Cola *colaptr, char * nomtoken);`

Esta función se utiliza para crear nuevos trabajos. Implementa la operación *crear* analizada anteriormente. *Crear* introduce por primera vez un trabajo al flujo del sistema. Tiene como parámetros el handle de la cola en la que se desea agregar el trabajo y el nombre del token a crear. Devuelve un código de error de tipo entero.

```
colaptr: Puntero a Cola que indica la cola a la que se
         desea agregar un trabajo nuevo para el sistema.
         Parámetro de Entrada.
nomtoken: String donde se especifica nombre del trabajo a
         crear. Parámetro de Entrada.
rc      : Entero que contiene el resultado de la operación.
```

`rc = TomarToken(IdType id, Cola *colaptr, char * nomtoken);`

Esta función solicita un nuevo trabajo a los servidores. Es la función que implementa la operación *tomar* analizada anteriormente. Tiene como parámetros al identificador (*id*) que fue devuelto por la función Present, el handle de la cola de la que desea obtener un trabajo y el nombre del trabajo a tomar. Si el tercer parámetro es vacío, significando que no se pide ningún trabajo en particular, pide el primer trabajo libre y retorna el nombre del token otorgado en el mismo parámetro. Si se especifica el niombre de un trabajo, toma el token especificado por nomtoken. Devuelve un código de error de tipo entero.

```
id      : IdType que identifica al cliente. Parámetro de
          Entrada.
Colaptr: Cola que indica la cola de la que se desea obtener
          un trabajo. Parámetro de Entrada.
```

```
Nomtoken:Puntero a carácter donde la función lee/retorna el
         nombre del trabajo. Parámetro de Entrada/Salida.
rc      : Entero que contiene el resultado de la operación.
```

`rc = PasarToken(IdType id, Cola *colaorigen, Cola *coladest, char *nomtoken);`

Esta función notifica que se terminó de trabajar con el trabajo *nomtoken* y lo pasa de la cola *colaorigen* a la *coladest*. Es la función que implementa la operación *poner* analizada anteriormente que habilita a los managers a entregar ese trabajo a algún proceso que lo solicite en la cola *coladest*.

Tiene como parámetros el identificador (*id*) que fue devuelto por la función Present, el handle de la cola a la que pertenece actualmente el trabajo, el handle de la cola a la que se desea pasar el trabajo ya terminado y el nombre del trabajo. Devuelve un código de error de tipo entero.

```
id      : IdType que identifica al cliente. Parámetro de
          Entrada.
colaorigen: Puntero a Cola que indica la cola a la que
          pertenece el trabajo. Parámetro de Entrada.
coladest: Puntero a Cola que indica la cola a la que se
          desea pasar el trabajo. Parámetro de Entrada.
nomtoken: Puntero a carácter donde se especifica el nombre
          de trabajo que se desea pasar. Parámetro de
          Entrada.
rc      : Entero que contiene el resultado de la operación.
```

`rc = BorrarToken(IdType id, Cola *colaptr, char *nomtoken);`

Esta función elimina un trabajo del flujo del sistema. Es la función que implementa la operación *borrar* analizada anteriormente. Tiene como parámetros al identificador (*id*) que fue devuelto por la función Present, el handle de la cola a la que pertenece el trabajo y el nombre del token a borrar. Devuelve un código de error de tipo entero.

```
id      : IdType que identifica al cliente. Parámetro de
          Entrada.
colaptr: Cola que indica la cola a la que pertenece el
          trabajo. Parámetro de Entrada.
nomtoken: Puntero a carácter que especifica el nombre del
          trabajo. Parámetro de Entrada.
rc      : Entero que contiene el resultado de la operación.
```

`printf(" Al presentarme : %s\n", DescribirError());`

Esta función devuelve un string que explica la naturaleza del último error ocurrido. Se puede utilizar para armar un mensaje de error a quien monitoree el sistema. Se retorna un string para que se pueda manejar como a la aplicación más le convenga desde cualquier plataforma soportada. (Presentation Manager de OS/2, AIX).

Como relacionar el concepto de trabajo y archivo asociado

Cada trabajo entregado por los managers para ser procesado por un cliente tiene asociado un archivo de igual nombre donde se encuentran los datos del trabajo. Este archivo está en el directorio cuyo camino completo asociado es devuelto por la función ConectarCola. Cada cola posee un filesystem asociado. Cada vez que se desee trabajar con los datos se deberán leer/escribir en el directorio correspondiente a la cola del trabajo.

Es muy importante respetar el orden de los pasos referidos a manipulación de trabajos y sus archivos asociados. Seguir las siguientes reglas:

- Crear el archivo de datos y escribirlo en el directorio correspondiente a su cola asociada antes de crear el token mediante la función CrearToken .
- Solicitar a los managers un trabajo mediante la función TomarToken y recién después proceder a leerlo desde el directorio de datos.
- Si se desea ejecutar la función PasarToken es porque se ha realizado un TomarToken, se ha leído su archivo desde el directorio y se ha transformado en otro archivo que se debe escribir en el directorio de la cola de datos correspondiente. Se debe primero escribir la salida en el directorio asociado, luego ejecutar la función PasarToken y por último borrar los datos de entrada que ya no serán necesarios al sistema. Si no se respeta este orden de ejecución se da la posibilidad de pérdida de datos.
- Ejecutar la función BorrarToken y recién después borrar físicamente el archivo de datos asociado al trabajo eliminado.

Todas estas guías se harán más claras en el ejemplo de utilización de las funciones de las APIs.

Ejemplo de un cliente general para estas APIs:

```
/* Includes necesarios para los clientes */
/* Headers del sistema */
# include <rpc/rpc.h>

/* Headers de las librerias */
#include "connect.h"
#include "cliente.h"
#include "errores.h"
#include "descrerror.h"


strcpy(tipoproc,"ocrproc"); /* ocrproc es un tipo de proceso definido*/
/*-------------------------------------------------------------------
   */
/* Presentarse al Manager como nuevo Cliente */

if (( self_id = Present(tipoproc, &myself)) < OK) {
   printf(" Al presentarme : %s\n", DescribirError());
   exit(1);
}



/*-------------------------------------------------------------------
   */
/* Conectarse a las colas de entrada y salida */

Ocr = ConectarCola(myself, "ocr", ocrdir);
if (Ocr == NULL){
   printf(" Al conectarse : %s\n", DescribirError());
   exit(1);
}

Fix1 = ConectarCola(myself, "fix1", fix1dir);
if (Fix1 == NULL){
   printf(" Al conectarse : %s\n", DescribirError());
   exit(1);
}



/*-------------------------------------------------------------------
   */
/* Ejemplo para Crear Tokens */

   < PROCESA >

   ....

   /* Un vez generado el job hay que salvar fisicamente el archivo que contiene
   los datos en el path indicado al conectarse con la cola */

   < ESCRIBE ARCHIVO DE SALIDA ASOCIADO AL TOKEN >

   if(( rc = CrearToken (Ocr, job)) != OK )
      printf(" Al crear : %s\n", DescribirError());
```

```
/*-------------------------------------------------------------------
   */
/* Ejemplo para TomarTokens y PasarTokens una vez procesados */

while (quiero_procesar_tokens) {
{
   /* Request de un trabajo */
   if(( rc = TomarToken(myself, Fix1, job)) != OK )
      printf(" Al tomar un trabajo : %s\n", DescribirError());

   /* Para leer el archivo de entrada hay que buscarlo en el directorio
   asociado a la cola,en este caso el contenido en la variable fix1dir
   */
      < LEE ARCHIVO DE ENTRADA ASOCIADO AL JOB >
   .....

   < PROCESA ARCHIVO DE ENTRADA PARA GENERAR LA SALIDA>

   ....

   /* Un vez generada la salida hay que salvarla fisicamente */
      < ESCRIBE ARCHIVO DE SALIDA ASOCIADO AL JOB >

   if(( rc = PasarToken(myself,Ocr, Fix1, job)) != OK )
      printf(" Al pasar un trabajo : %s\n", DescribirError());
   else
      /* En el directorio asociado a la cola de entrada. Ahora: ocrdir
   */
      < BORRA ARCHIVO DE ENTRADA  ASOCIADO AL JOB  >


} /* End While */

/*-------------------------------------------------------------------
   */
/* Ejemplo para BorrarTokens */

   if(( rc = BorrarToken(myself, Ocr, job)) != OK )
      printf(" Al borrar un trabajo : %s\n", DescribirError());
   else
      /* En el directorio asociado a la cola de entrada. Ahora: ocrdir
   */
      < BORRA ARCHIVO ASOCIADO AL JOB  >
```

Es importante usar los pasos antes mencionados en el orden dado.

El proceso escrito por el usuario es el encargado de borrar los archivos de entrada usados para su proceso una vez finalizado el mismo. Solo deben borrarse DESPUES de ejecutar satisfactoriamente el *PasarToken*.

Los posibles errores son:

Código de Error :

/* Errores que devuelven los servers a los procesos clientes */

ERROR_COLA_NODEFINIDA : Error al solicitar una operación sobre una cola inexistente.

ERROR_NO_PERMISO: Error al solicitar una operación no permitida sobre la cola especificada.

PROC_NO_REG: Tipo de proceso no definido.

NO_SPACE: No hay mas espacio en las tablas del server. No se puede realizar esta operación.

ID_INCORRECTO: El id utilizado no es correcto.

NO_EXISTE: El token especificado no existe.

/* Errores que devuelve el watchdog (o su interface con el cliente) al cliente. */

ERROR_WATCHDOG: El watchdog tiene problemas. Este debe estar corriendo en la misma máquina del cliente para poder ejecutar éste.

ERROR_MONITOR: El monitor tiene problemas. Este debe estar corriendo para comenzar algún cliente.

/* Errores producidos al solicitar una función algún server y no obtener respuesta */

ERROR_CLNT_CREATE: No pudo crearse el cliente de RPC para comunicarse con los servers.

ERROR_COMUNICACION: Problemas en la comunicación con el server.

7.4 Segunda parte

La segunda parte de este manual, destinada al administrador de la herramienta, no se incluye en el presente trabajo de Tesis de Licenciatura por tratarse de una componente netamente relacionada a la implementación de un proyecto. Por la existencia de esta sección se pretende hacer notar la importancia de la misma para la completitud de la documentación de una herramienta con los objetivos y fines descriptos en este trabajo.

8. Anexo A: Características de los sistemas distribuidos

El objetivo de la presente sección es presentar un apéndice general referido a los sistemas distribuidos.

En la sección 4.3 "Consideraciones para el diseño de la herramienta", página 32 del presente trabajo de licenciatura se hizo referencia a algunas características comunes a los llamados sistemas distribuidos [Tel 001], [Mullender 002], [Singhal 003], [Chandy 005] [Tanenbaum 006 007 008], que repetiremos a continuación:

- Los procesos corren en varias computadoras que no comparten memoria ni un reloj global. Cada computadora tiene su propia memoria y corre su propio sistema operativo. Comparten un protocolo de comunicación común.

- Las computadoras se comunican entre ellas intercambiando mensajes a través de una red de comunicación.

- Los recursos propios de una computadora y manejados por ésta se dicen locales, mientras que los de otras se dicen remotos. Generalmente acceder a recursos remotos es más costoso que acceder a recursos locales debido a los retrasos provocados por la red de comunicación y el overhead de CPU causado por procesar los protocolos de comunicación.

- Falta de Conocimiento global: Cada proceso del sistema no tiene un conocimiento global del estado del resto del sistema en forma completa y correcta.

- Falta de Global Naming: Los nombres se utilizan para referenciarse a los objetos del sistema. Los procesos no conocen como referenciar a otros procesos del sistema, ni siquiera pueden conocer su existencia.

- Escalabilidad: El sistema distribuido de procesos que nos interesa puede crecer con el tiempo. El diseño de la herramienta debe favorecer este hecho sin degradar la performance.

- Compatibilidad: se refiere a la noción de interoperabilidad de recursos dentro del sistema. Se podrían nombrar tres niveles de compatibilidad en sistemas distribuidos: binario, ejecución, protocolo.

  En un sistema compatible a nivel binario todos los sistemas ejecutan el mismo conjunto de instrucciones binarias, aunque los procesadores puedan diferir en performance y input-output.

- *Interconexiones*: Algunas de las conexiones de entrada/salida, interconectarán a las computadoras entre sí. Para que se trate de un sistema distribuido, las mismas deben ser capaces de intercomunicarse.

- *Estado compartido*: Las computadores cooperan para mantener alguna forma de estado compartido. Si la correcta operación del sistema se describe en función de algunos invariantes globales, por lo tanto, el mantenimiento de dichos invariantes requiere la correcta y coordinada operación de las múltiples computadoras.

Construir un sistema a partir de computadoras interconectadas, requiere que se consideren cuatro importantes puntos:

- *Falla independiente*: Como hay varias computadoras, al fallar una de ellas las demás deben ser capaces de seguir procesando. El "sistema" debe continuar trabajando aunque una o más fallen.

- *Comunicación no confiable*: En la mayoría de los casos las conexiones entre las computadoras no se encuentran en un entorno debidamente controlado, lo que provoca que no funcionen adecuadamente todo el tiempo. Las conexiones pueden no estar disponibles. Los mensajes pueden perderse o corromperse. Una computadora no puede contar con ser capaz de comunicarse claramente con otra, aunque las dos funcionen correctamente.

- *Comunicación insegura:* Las interconexiones entre las computadores pueden estar expuestas a monitoreo no autorizado o a modificación de los mensajes.

- *Comunicación costosa:* La interconexión entre las computadoras generalmente proporciona menor ancho de banda, mayor latencia, y mayor costo de comunicación que la disponible entre procesos independientes dentro de una misma máquina.

Un sistema centralizado que soporta múltiples procesos y provee alguna forma de comunicación entre procesos, como un sistema timesharing Unix, puede mostrar en forma virtual las tres características primarias de un sistema distribuido. Hasta puede mostrar falla independiente: el demonio de correo falla sin pertubar a otros procesos independientes del mismo. El diseño y las técnicas asociadas a la comunicación de procesos secuenciales en sistemas centralizados forman parte de las técnicas básicas en sistemas distribuidos. Sin embargo, la mayoría de los sistemas centralizados son exitosos sin necesidad de lidiar con falla independiente y generalmente no se enfrentan a comunicación no confiable, insegura y costosa. En sistemas distribuidos estos puntos siempre deben resolverse.

El ejemplo canónico de un sistema distribuido actual es un conjunto de estaciones de trabajo interconectadas a través de una red.

8.1.1 Sistemas centralizados versus sistemas en red

Los sistemas basados en red permiten compartir información y recursos diseminados geográfica y organizacionalmente. Permiten el uso de computadoras con buena razón de costo-beneficio y acercan el ciclo de cómputo a los datos. Pueden crecer en pequeños incrementos a través de un amplio rango de tamaños. Permiten autonomía mediante decisión de compra de diferentes componentes, múltiples proveedores, versiones de software, y adopción de múltiples políticas de administración. No necesariamente fallan simultanéamente. Por lo tanto, en las áreas de costo, compartir recursos, crecimiento, y autonomía son mejores que los sistemas centralizados tradicionales.

Por otro lado, los sistemas centralizados hacen algunas cosas en forma más sencilla. Toda la información y recursos en estos sistemas son igualmente accesibles. Las funciones trabajan de igual manera y los objetos tienen el mismo nombre en todo el sistema. Son más fáciles de administrar. Son, en definitiva, más accesibles, coherentes y administrables.

En las áreas de seguridad y disponibilidad la comparación no proporciona una ventaja especial de uno sobre el otro.

En la práctica, hoy en día, ni los sistemas centralizados ni los distribuidos sobre una red proveen seguridad, pero por diferentes razones. Un sistema centralizado tiene un único dominio de seguridad bajo una única autoridad. La base computacional esta contenida en un único y confiable sistema operativo que ejecuta en una única máquina que puede tener un ambiente físico seguro. Toda la seguridad está centralizada, es más fácil de monitorear y mantener, todos saben a quien referirse ante un problema. Por otro lado, es difícil eliminar todas las fallas de un sistema operativo, o el entorno operativo y con un único dominio de seguridad, una sola falla permite el acceso indebido a todo el sistema. Los sistemas basados en red tienen múltiples dominios de seguridad, y demuestran los problemas inversos de seguridad. La base computacional confiable ejecuta en componentes que operan en entonos con variados niveles de seguridad física, políticas y autoridades. Las interconexiones entre las computadoras son generalmente inseguras. Es difícil saber qué es confiable y en qué se puede confiar. Pero, como el sistema contiene múltiples computadoras, explotar una falla de seguridad en una de ellas, sólo da acceso a la misma y no al sistema entero.

Con respecto a la disponibilidad, también puede hacerse un estudio de este tipo. Un sistema centralizado puede tener un entorno físico y operacional controlado. Como muchas fallas son resultado de factores físicos u operacionales, una administración minuciosa puede proveer buena disponibilidad. Pero cuando algo falla, el sistema completo falla. En un ambiente en red, varias computadoras fallan independientemente. Sin embargo, a menudo es necesario que varias máquinas funcionen simultáneamente para llevar a cabo una tarea. Por esto, la probabilidad de falla es mayor que la probabilidad de falla de cada uno de los componentes. Cuando la falla independiente es debidamente manejada a través de replicación de funciones y componentes, múltiples fallas son necesarias antes de dañar la disponibilidad o confiabilidad del sistema. También deben manejarse las fallas de comunicación, que no sólo llevan a no

Compatibilidad a nivel de ejecución existe si un mismo código fuente puede ser recompilado y ejecutado en cualquiera de las máquinas del sistema.

La compatibilidad a nivel de protocolo es la menos restrictiva. Logra interoperabilidad al requerir que todos los sistemas soporten un conjunto común de protocolos. Así se logra que cada computadora pueda ejecutar su propio sistema operativo sin perder interoperabilidad.

- Sincronización de procesos: Surge la necesidad de sincronizar a los procesos que desean acceder concurrentemente a un recurso común. Hay que prevenirse contra problemas como deadlock, inanición, etc.

- Aunque existen implementados en la actualidad numerosos sistemas operativos distribuidos, el presente Trabajo de Tesis está orientado a aquellos entornos donde no se cuenta con la disponibilidad de alguno de estos. Se pretende suplir las características que poseen los mismos y que fueran necesarias para la solución del problema planteado.

A continuación se amplia la información presentando algunas características, ventajas y desventajas de los sistemas distribuidos. No se discuten sistemas distribuidos específicos para determinadas aplicaciones, como sistemas bancarios automatizados, sistemas de control para roaming y seguimiento de teléfonos celulares o puntos de venta para aplicaciones de mayoreo (retail) o muchos otros en uso actualmente. Sin embargo, los puntos mencionados en esta sección aplican a muchos de dichos sistemas. Tampoco se cubren en esta sección sistemas distribuidos de tiempo real, como sistemas de control para fábricas, aviones, o automóviles, que enfrentan requerimientos únicos de planificación de tiempos y utilización de recursos que no coinciden con los sistemas a los cuales aplica el presente trabajo de tesis.

8.1 Características generales de sistemas distribuidos.

Un sistema distribuido es un conjunto de computadoras realizando una tarea en conjunto. Por lo tanto, un sistema distribuido tiene tres características primarias.

- *Múltiples computadoras*: Un sistema distribuido tiene más de una computadora física, cada una consistente de uno o varios procesadores, memoria local, posiblemente almacenamiento estable como ser discos, o conexiones de entrada/salida conectándola al ambiente.

disponibilidad, sino también a funcionamiento incorrecto. Una máquina no puede distinguir un vecino que falla de uno que está incomunicado. Mantener invariantes de estado global en estos casos es difícil. Se requiere un diseño detallado y cuidadoso para obtener funcionamiento correcto y disponibilidad de los sistemas a través de la replicación.

8.1.1.1 Ventajas de Sistemas Distribuídos por sobre Sistemas Centralizados

Existen numerosas, y de diferente índole, ventajas de los sistemas distribuidos sobre los centralizados.

- ***Economía***: Los sistemas distribuidos tienen, en general, una mayor razón precio/performance, que los sistemas centralizados que cumplen funciones comparables.

- ***Velocidad***: Algunos hacen una distinción entre sistemas distribuidos, cuyo objetivo es permitir que varios usuarios trabajen al mismo tiempo, y sistemas paralelos, cuyo único objetivo es lograr una máxima velocidad en la resolución de un determinado problema.

- ***Distribución inherente***: Algunas aplicaciones están ya pensadas para ocupar máquinas separadas. Muchos procesos manuales existentes deben ser automatizados. Los mismos están, generalmente, pensados para distribuir trabajo a distintas personas y posibilitar que el conjunto de la tarea se divida en etapas o fases distribuidas en el tiempo. Este estilo de proceso es más fácilmente automatizado a un sistema distribuido que a uno centralizado.

- ***Confiabilidad y Disponibilidad***: Si se cae una de las máquinas de un sistema distribuido, el sistema como un todo puede todavía seguir funcionando -siempre que se haya diseñado a tal efecto-. El objetivo es que luego de la caída de uno de los componentes de un sistema distribuido, se puede redireccionar el trabajo que estaba realizando un componente que falló a otro componente. La disponibilidad es la cantidad de tiempo que el sistema se encuentra en estado utilizable. Se utiliza redundancia en los componentes de hardware y software para aumentarla. Un sistema altamente confiable debe ser altamente disponible, sin ser esto suficiente. Los datos no deben perderse y deben ser consistentes. En general, a mayor cantidad de copias, mayor disponibilidad pero asimismo, mayor posibilidad de inconsistencias.

- ***Escalabilidad***: Se pueden ir agregando más componentes de software y hardware con el objetivo de incrementar el poder computacional total del sistema distribuído.

8.1.1.2 Ventajas de Sistemas Distribuídos por sobre PCs independientes

- ***Compartir datos***: Permite que varios usuarios accedan a bases de datos y estructuras de datos comunes.
- ***Compartir recursos*** : Permite compartir periféricos caros entre los usuarios.
- ***Comunicación*** : Hace que la comunicación entre usuarios sea más sencilla.
- ***Flexibilidad*** : Permite distribuir el trabajo a través de las máquinas que haya disponibles para ser usadas, permitiendo variar dinámicamente la carga asignada a cada una de acuerdo a parámetros de performance aceptables o preestablecidos.

8.1.1.3 Desventajas de los Sistemas Distribuídos

- ***Software:*** Existe mucha diversidad y cantidad de software para este tipo de sistemas, sin haber un estándar. Por ejemplo: los sistemas de monitoreo de sistemas y red son más dificiles y caros de implementar en sistemas distribuidos.
- ***Networking***: La red es fuente de numerosos problemas si no es adecuadamente mantenida y monitoreada para soportar los cambios del sistema. De acuerdo a la topología y tecnología de la red, puede saturarse, tener problemas de colisiones, performance, etc.
- ***Seguridad :*** Hay mayor facilidad de acceso a datos confidenciales. Las medidas de seguridad deben ser debidamente diseñadas y controladas.

8.2 Miscelaneos sobre sistemas distribuidos

En la presente sección se introducen de manera general conceptos normalmente relacionados con los sistemas distribuidos. Los temas aquí incluidos fueron elegidos por estar relacionados con el presente trabajo de tesis y mencionados o considerados en el desarrollo de la misma.

8.2.1 Transparencia

Se dice que un sistema es transparente cuando logra dar una imagen única del sistema, como si fuera un sistema centralizado. La transparencia se puede lograr a dos niveles diferentes. La más sencilla es esconder la distribución a los usuarios. Por ejemplo

si se compila un conjunto de programas, no hay necesidad que el usuario se entere si se está compilando paralelamente en computadoras distintas accediendo a varios servidores de archivos. En un nivel más profundo y más difícil, es posible hacer que el sistema sea transparente a los procesos. Debe diseñarse la interface de la llamada al sistema para ocultar la existencia de varios sistemas.

El concepto de transparencia puede aplicarse a distintos aspectos de los sistemas distribuidos.

- ***Transparencia de locación:*** se refiere al hecho que en un sistema realmente distribuido el usuario no puede decir en qué lugar se encuentran los recursos que está utilizando. El nombre del recurso no debería incluir la localización del mismo que se muestra en el siguiente ejemplo *máquina1:programaA*.

- ***Transparencia de migración:*** significa que los recursos pueden cambiar su ubicación sin cambiar su nombre.

- ***Transparencia de concurrencia:*** los sistemas distribuidos tienen usuarios concurrentes e independientes. Si dos usuarios tratan de actualizar el mismo archivo al mismo tiempo, el sistema debe ser capaz de proveer mecanismos para lograrlo sin perjudicar la consistenica de los mismos. Por ejemplo, que se lockee un archivo cuando un usuario comienza a utilizarlo y deslockearlo cuando haya terminado el acceso.

- ***Transparencia de paralelismo:*** representa la más difícil de alcanzar. Si un programador está desarrollando un sistema y quisiera hacer un uso eficiente de todos los recursos de los cuales dispone en forma paralela, lo deseable sería que el compilador, como el sistema de ejecución, como el sistema operativo deberían resolver este problema en forma autónoma eficientemente sin que el programador tenga que enterarse. Desafortunadamente, se está muy lejos de alcanzar este objetivo.

8.2.2 Comunicación en Sistemas Distribuidos.

La mayor y más importante diferencia en un sistema distribuido es el mecanismo de comunicación interprocesos.

En caso de un único sistema, se asume la existencia de memoria compartida. Hasta la sincronización de dos procesos requiere que haya un área en común donde se va a alojar la variable de semáforo. En un sistema distribuido esto no es posible. Las llamadas a procedimientos remotos -RPC: Remote Procedures Calls-, son una manera fácil de empaquetar el pasaje de mensajes y hacerlo ver como si fuera parte de la programación convencional. Para mayores detalles referirse al "Anexo C: Llamada a Procedimiento Remoto - Remote Procedure Call".

Debido a la falta de memoria compartida, toda comunicación se basa en el pasaje de parámetros. Cuando un proceso A se quiere comunicar con otro B, arma un mensaje en su propio espacio de direccionamiento, luego ejecuta una llamada al sistema operativo, quien toma el mensaje y lo envía a través de la red al proceso B. Esto suena bastante sencillo pero no lo es. Hay que tener en cuenta que A y B se tienen que poner de acuerdo primero en el protocolo de intercambio de información que usarán. El mismo incluye si los bits se toman de derecha a izquierda o viceversa, si se mandan las cosas en EBCDIC y lo espera en ASCII y mucho más.

Para manejar todos esto detalles, la ISO (International Standards Organization) ha desarrollado un modelo de referencia, que identifica claramente los niveles, los nombres que tienen y qué tipo de trabajo se realiza en cada uno de ellos. Este modelo se llama OSI Open Systems Interconnection Reference Model. Se dice que un sistema es abierto cuando está preparado para comunicarse con cualquier otro sistema abierto mediante las reglas estáandar que rigen formato, contenido, y semántica de los mensajes enviados y recibidos. Estas reglas están formalizadas y se las llama protocolos. Básicamente un protocolo es un acuerdo de cómo debe ser llevada a cabo la comunicación. La OSI distingue entre dos tipos de protocolos, *protocolos con conexión* que establecen una conexión explícita entre el que envia y recibe y posiblemente negocian qué protocolo van a utilizar, antes de intercambiar datos. Cuando terminan de hacer la transferencia, la conexión se da de baja explícitamente. El teléfono es el clásico ejemplo de comunicación orientada a conexión. Con *protocolos sin conexión*, no se realiza ningun tipo de negociación previa, directamente se envían los datos cuando el que los envia está listo. Ejemplo de esto es el sistema de correo, poner una carta en un buzón.

Referirse a [008] [TAN/91] Tanenbaum, A. *Redes de Ordenadores*, para mayores detalles.

8.2.3 NFS (Network File System)

Así como los términos fuerte y débilmente acoplado se aplican al hardware, también pueden usarse para el software. El software débilmente acoplado permite a las máquinas y usuarios de sistemas distribuidos ser, fundamentalmente, independientes unos de otros, pero interactuar en un nivel limitado cuando es necesario. Podría hablarse de software fuertemente acoplado al que permite a un sistema multiprocesador correr un programa de juego de ajedrez en paralelo. A cada procesador se le asigna un tablero para evaluar y cuando termina de procesar todos los tableros posibles generados por éste, continua evaluando un nuevo tablero. Este software es claramente más fuertemente acoplado que el anterior. De esta manera combinando distintos tipos de hardware y software se obtienen sistemas con diferentes grados de acoplamiento. Presentamos a acontinuación algunos ejemplos.

En este caso se ejemplifica el uso de sistemas de hardware débilmente acoplados con software débilmente acoplado: una serie de estaciones de trabajo independientes

conectadas a través de una red de área local. Para trabajar en una máquina remota bastaría con ejecutar comandos del estilo ***rlogin machine*** que nos permitirían transformar nuestra propia estación de trabajo en una terminal de la máquina a la que nos acabamos de conectar remotamente. Todo comando tipeado se transfiere a la máquina remota, y todo resultado generado por la máquina remota se muestra en la primer terminal.

Asimismo, si existiera la necesidad de copiar archivos entre dos máquinas distintas se pueden efectuar comandos como ***rcp machine1:file1 machine2:file2*** . En este caso el usuario debe conocer en qué máquina quiere copiar el archivo.

Este tipo de comunicación es un poco primitiva y ha llevado a los diseñadores a buscar nuevas y más transparentes alternativas. Una solución es proveer un sistema de archivos global, que pueda ser accedido por todas las estaciones de trabajo.

Los servidores de archivos generalmente mantienen un sistema de archivos jerárquico, y cada usuario puede importar o montar estos filesystems, aumentando su capacidad local. Los sistemas operativos utilizados en estos entornos deben proveer manejo de recursos locales y remotos y la comunicación entre ellos. No es imprescidible que todas las máquinas tengan el mismo sistema operativo para poder lograr esta comunicación. Lo mínimo que tienen que tener es un acuerdo sobre los protocolos de comunicación que utilizarán, el formato y la semántica de los mensajes que van a intercambiar. El más conocido es el **Network File System**, conocido mundialmente como **NFS**, de Sun Microsystems. El NFS fue originalmente desarrollado para ser utilizado en estaciones de trabajo Unix, pero actualmente soporta sistemas heterogéneos.

Hay tres aspectos interesantes del NFS: su arquitectura, el protocolo, y su implementación. En esta sección los mencionaremos superficialmente por su contribución a la forma de compartir datos en sistemas distribuidos y en particular en el presente trabajo de Tesis de Licenciatura.

Arquitectura de NFS

La idea básica es lograr que usuarios y estaciones de trabajo heterogéneas puedan compartir un mismo filesystem. Muchas veces tanto clientes como servidores se encuentran en la misma LAN pero pueden montarse filesystems a través de una WAN, o en la misma máquina.

Cada servidor de NFS *exporta* uno o más directorios, haciendo también accesibles todos sus subdirectorios. La lista de directorios exportables de un servidor permite que los mismos sean exportados automáticamente cada vez que se bootea el mismo.

Los clientes, acceden a estos directorios a través del comando *mount*. Si la estación de trabajo no tiene disco, directamente se puede montar como raíz cualquier directorio remoto. Sino, puede ser montado en cualquier directorio local. Para aquellas aplicaciones corriendo en las estaciones de trabajo es totalmente transparente si los archivos a los cuales accede son locales a la máquina o son remotos. Si dos estaciones de trabajo montan el mismo filesystem ambas pueden comunicarse compartiendo archivos en subdirectorios comunes, ya que ambas ven lo mismo como si fuera local. Los archivos compartidos se encuentran en las jerarquías de directorios de varias máquinas, y son accedidos de la misma manera que la usual.

Protocolos NFS

Como uno de los objetivos de NFS es soportar sistemas heterogéneos, con clientes corriendo en diferentes sistemas operativos, en diferentes hardware, es importantísimo tener una interfaz entre clientes y servidores muy bien definida.

Esto se logra a través de dos protocolos cliente-servidor. Un ***protocolo*** es un conjunto de requerimientos enviados por los clientes a los servidores. Así los servidores pueden reconocer los pedidos y llevarlos a cabo, sin necesidad de saber nada, absolutamente nada de quién es ni que características tiene el cliente que está realizando dicho requerimiento. Similarmente, los clientes pueden tratar a los servidores como "cajas negras" que aceptan y procesan un conjunto específico de requerimientos.

El primer protocolo maneja el montado. El cliente envia un nombre de directorio al servidor requiriendo permiso para montarlo. Si el mismo existe, está exportando y los permisos son adecuados, el servidor retorna un manejador de archivo que será usado por el cliente en subsiguientes accesos. El servidor no necesita saber en qué directorio el cliente montará el filesystem.

El segundo protocolo es para el acceso a archivos y directorios. Los clientes envian mensajes a los servidores para manipular directorios, leer y escribir archivos, así como muchos otras llamadas al sistema. NFS tiene su propio mecanismo para manejar lockeo de archivos.

Implementación de NFS

La implementación del código es independiente de los protocolos. Estudiaremos superficialmente la implementación de Sun Microsystems que consta de tres capas. La primera, es el nivel de llamadas al sistema, que manejan pedidos como *open, close, read.* Una vez parseada la llamada y chequeados sus parámetros, se llama a una segunda capa, el VFS Virtual File System. Su función es la de mantener una tabla con una entrada por cada archivo abierto, v-nodes (análogo a la tabla de i-nodos de Unix). Mantiene información sobre si un archivo es local o remoto, y demás información necesaria para poder accederlo.

Un ejemplo del funcionamiento sería: si una máquina ejecuta un comando de *mount o: eolo:/usr/mydir*, cuando se parsea este comando, se encuentra el nombre de la máquina remota (*eolo*) entonces se comunica con ésta y averigua si realmente existe un directorio bajo el nombre de /usr/mydir y si éste está como exportable. Si se cumplen estos requisitos, el servidor retorna un manejador de archivo para ese directorio. Finalmente, el servidor ejecuta un *mount* para pasarle este manejador al kernel.

El kernel entonces construye un v-node para este directorio remoto, y pide al código NFS del cliente que construya a su vez un r-node (remote i-node) en sus tablas internas para contener a este manejador de archivo. Entonces el v-node apunta al r-node. Cada v-node de la capa de VFS va a contener un puntero al r-node del código NFS del cliente o un puntero a un i-nodo local. Ya que desde el v-node es posible ver si un archivo o un directorio son locales a la máquina o remotos, y si lo son, el manejador de archivo para accederlos.

Cada vez que se abre un archivo remoto, el kernel se da cuenta que el directorio donde está buscando los datos es un directorio remoto, entonces en el v-nodo del directorio va a encontrar el puntero al r-nodo y le pide al cliente NFS que abra el archivo. El cliente NFS busca el resto del camino para llegar al archivo y trae un manejador de archivo para éste. Arma un r-node para el archivo remoto en su tabla local, y lo reporta a la capa de VFS, quien se encarga de poner un v-node para el archivo que se quiere acceder apuntando al r-node que se acababa de armar. Se sigue el concepto de que cada v-node apunta a un r-node o a un i-node (remoto o local, respectivamente). Notar que no se mantienen ningún tipo de tablas del lado del servidor, sólo del lado del cliente. Aunque el servidor esté preparado para devolver manejadores de archivo cada vez que se lo requiere, éste no lleva un registro de a quién le dio cada manejador de archivo.

Por eficiencia, transferencias entre clientes y servidor se hacen con dimensiones de buffer bastantes grandes, aunque lo que se haya pedido sea menor.

NFS ha sido criticado por no implementar la semántica de Unix. Una escritura de un archivo en un cliente, puede o no verse por otro cliente dependiendo del momento de la lectura. Sin embargo, NFS es ampliamente usado como mecanismo para compartir archivos.

9. Anexo B: Semáforos en Unix

Las llamadas a sistema de semáforos permiten a los procesos manejar conjuntos de semáforos y realizar operaciones atómicas sobre estos conjuntos.

9.1 Operaciones sobre los semáforos

Unix soporta las siguientes llamdas al sistema: semget, semop, semctl y semipcs que se explicarán a continuación.

9.1.1 semget (key, nsems, flag)

Esta llamada al sistema permite crear un conjunto de semáforos o compartir uno ya existente.

- Key es la clave con que el proceso identifica a un conjunto de semáforos. De esta manera, dos procesos que quieran compartir un conjunto de semáforos, deberán hacer un semget con el mismo key.

- Nsems es la cantidad de semáforos que tiene el conjunto. Cuando el conjunto ya existe este parámetro puede ser 0 (cero) o un número menor al especificado al crear el semáforo.

- El valor de semflag es un entero que especifica los permisos para las operaciones y campos de control IPC_CREAT y IPC_EXCL. El primero indica que se desea crear un nuevo conjunto de semáforos si el solicitado no existe y el segundo indica al system call que le retorne error si el semid ya existe para la key dada.

Si ya existe la clave key en la estructura de semáforos, significa que el proceso desea compartir con otro proceso ese conjunto de semáforos. Por este motivo se le devuelve el mismo identificador de semáforos (descriptor) que el proceso que lo creó. En caso de no existir la clave, se devuelve un nuevo descriptor para un nuevo conjunto de semáforos.

Para realizar estas operaciones se deberán tener en cuenta varios aspectos. El primero es que existe un límite en la cantidad de conjuntos de semáforos que se pueden pedir. En caso de superar esta cantidad, se devolver error. El segundo aspecto es que para

poder crear un nuevo semáforo debo setear los flags de tal manera que al hacer un AND con la constante IPC_CREAT me de verdadero.

El caso en que se realiza un semget con el valor de key en IPC_PRIVATE significa que el proceso quiere siempre un nuevo conjunto de semáforos. Se devuelve error en caso de que la cantidad de conjuntos de semáforos existentes ya haya sido cubierta.

Resumen de las banderas para las operaciones:

Si key = IPC_PRIVATE => Se retorna un nuevo conjunto de semáforos.
Si key <>IPC_PRIVATE =>
  Si ya existe key =>
    Si flag est seteado con IPC_CREAT y con IPC_EXCLU => error.
    Si flag est seteado con IPC_CREAT y no con IPC_EXCLU =>
        devuelvo el conjunto de semáforos existente.
    Si flag no est seteado con IPC_CREAT (con o sin) IPC_EXCLU
        => devuelvo el conjunto de semáforos existente.

  Si no existe key =>
    Si flag est seteado con IPC_CREAT (con o sin) IPC_EXCLU =>
        devuelvo un nuevo conjunto de semáforos.
    Si flag no est seteado con IPC_CREAT (con o sin) IPC_EXCLU
        => error.

9.1.2 semop(semid,sops,nsops)

Esta llamado al sistema permite realizar operaciones sobre cada uno de los semáforos de un conjunto.

- Semid indica el conjunto de semáforos sobre el cual se realizarán las operaciones.

- Sops es un puntero a un array ubicado en memoria del usuario. Este array contiene, en cada posición, el número del semáforo sobre el que se realizará la operación (sem_num), la operación a realizar (sem_op) y flags de control (sem_flg).

- Nsops es un entero indicando la cantidad de posiciones del array apuntado por sops.

Para poder realizar cualquier operación se debe verificar que el semáforo haya sido creado, es decir, que el proceso haya hecho un semget sobre el conjunto de semáforos identificado por semid. También se verifica si el proceso tiene los permisos necesarios para realizar las operaciones que desea. En caso contrario se retorna error de acceso, sin haber realizado ninguna operación, ni siquiera aquellas para las que tenía permiso.

Las operaciones a realizar están dadas por el valor de sem_op:

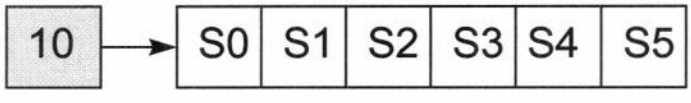
Si sem_op > 0 => sem_val = sem_val + sem_op
Si sem_op < 0 => sem_val = sem_val - sem_op
Si sem_op = 0 => se testea el semáforo por igual a cero

La primera operación equivale a hacer un V(sem_num) mientras que la segunda equivale a hacer un P(sem_num). En caso de querer hacer una operación que no puede ser satisfecha sobre alguno de los semáforos del array, se deberá hacer un roll-back sobre todo ese conjunto. Una operación no podrá ser satisfecha cuando se quiera hacer un P y no haya la suficiente cantidad de recursos o cuando se testee el semáforo por cero y sem_val sea distinto de cero.

Hacer un roll-back significa que todos los valores de los semáforos que fueron modificados al intentar realizar la operación deberán ser restablecidos al valor que tenían antes de intentar realizar la operación que no pudo ser satisfecha. Es por esto que decimos que todas las operaciones sobre los conjuntos de semáforos son atómicas.

Si un proceso llega a esta situación deberá bloquearse hasta que otro proceso realice alguna operación sobre el conjunto de semáforos que permita suponer que los procesos dormidos podrán cumplir las operaciones pedidas. Daremos un ejemplo para explicar por qué decimos que sólo podemos "suponer" que las operaciones realizadas por un proceso que despierten a proceso dormidos permitan efectivamente que los mismos completen sus operaciones.

Supongamos el siguiente array de semáforos:



y supongamos también dos procesos, P1 y P2 bloqueados en la cola del semáforo 10. P1 quedó bloqueado cuando intentó hacer un P sobre el semáforo S4 y P2 cuando intentó hacer un P sobre el semáforo S1. Ahora supongamos que un proceso P3 logra realizar satisfactoriamente todas sus operaciones; las que sólo realizaban un V sobre el semáforo S5. Como la única información con la que se cuenta es que hay dos procesos esperando por el conjunto 10 y que existe un proceso P3 que acaba de realizar

operaciones V sobre el conjunto 10, voy despertando uno a uno los procesos en la cola de incremento que esperan por el descriptor 10 para que intenten realizar sus operaciones. Ni P1 ni P2 tendrán éxito pues si bien P3 realizó operaciones V sobre el conjunto, no lo hizo particularmente sobre S1 o S4, los semáforos por los que esperaban P1 y P2.

Si en cambio suponemos que P3 incrementa el valor de S1 de tal manera que P2 pueda realizar su operación sobre ese semáforo, P2 podrá continuar intentando realizar sus restantes operaciones sobre los demás elementos del conjunto, las que pueden terminar exitosamente o no.

NOTA sobre la operación con código de operación sem_op < 0:

Al realizar una operación sem_op con código menor que cero se está intentando realizar un P sobre un determinado semáforo. Según lo expuesto arriba un proceso que desee ejecutar un sem_op con código -5, cuando el valor del semáforo es 3; podrá ejecutarlo resultando en el valor final del semáforo -2 y el proceso bloqueado.

Esta no es la semántica más intuitiva para la operación P sobre un semáforo. Lo que se quisiera reflejar es lo siguiente: si el proceso realiza un semop con código -5 se entiende que deberá bloquearse hasta tanto no haya obtenido 5 unidades del semáforo. Si antes de hacerlo decrementa el valor de semáforo (dejándolo en -2 en este caso), impedirá que otros procesos que desearan ejecutar semop y estuvieran en condiciones de hacerlo, lo hagan satisfactoriamente.

En el caso anterior si un proceso deseaba realizar un semop con código -1 podría haberlo hecho antes que se decrementara el semáforo, pero no luego. Por esto preferimos la versión donde primero se testea si el semáforo puede ser disminuido en la cantidad de unidades solicitadas en el código de semop, si este es el caso se decrementa el valor del semáforo y se finaliza el system call. En caso contrario se bloquea al proceso solicitante pero NO se decrementa el valor del semáforo.

```
Si sem_op < 0 entonces
  Si (sem_op + sem_val) < 0 entonces
    bloqueo al proceso solicitante
  sino
    sem_val = sem_val + sem_op
  finsi
finsi
```

Esta versión evita deadlocks aunque podría causar la inanición de un proceso que siempre solicita la operación semop con códigos negativos de gran valor absoluto. A este tipo de procesos se le antepondrían, en el caso de no poder satisfacerse su pedido inmediatamente, aquellos que solicitaran cantidades de valor absoluto más chicas.

La versión anterior donde se decrementaba el valor del semáforo independientemente si el pedido podía ser satisfecho en ese momento o no, trata de evitar inanición pues al decrementar el valor es una forma de impedir que otros procesos se cuelen indefinidamente.

9.1.3 semctl (semid,semnum,cmd,args)

Este system call permite realizar operaciones de control sobre el conjunto de semáforos identificado con semid y, en particular, el semáforo identificado con semnum. Arg es del tipo union de un entero, un puntero a la estructura de semáforos y un puntero a un entero sin signo.

El parámetro cmd puede tomar los siguientes valores:

Comando	Explicación	Valor de Arg
GETVAL	Devuelve el valor del semáforo semnum del conjunto indicado por semid	Ignorado
SETVAL	Setea el valor del semáforo semnum de conjunto indicado por semid	Arg.val de entrada
GETPIDS	Devuelve el PID del proceso que realizó la última operación sobre el semáforo semnum del conjunto indicado por semid.	Ignorado
GETNCNT	Devuelve la cantidad de procesos esperando que incremente el valor del semáforo semnum del conjunto indicado por semid.	Ignorado
GETZCNT	Devuelve la cantidad de procesos esperando que el valor del semáforo semnum del conjunto indicado por semid sea cero.	Ignorado
GETALL	Devuelve el valor de todos los semáforos del conjunto indicado por semid	Arg.array de salida
SETALL	Setea el valor de todos los semáforos del conjunto indicado por semid.	Arg.array de entrada
IPC_STAT	Devuelve la información de estado que está en la estructura del semid_ds, y la coloca en la estructura de datos apuntada por el puntero buf en la memoria del usuario. Arg.buf contiene este puntero	Arg.buf de salida
IPC_SET	Setea el uid y gid efectivos y los permisos para el conjunto de semáforos indicado	Arg.buf de entrada.

	por el semid.	
IPC_RMID	Se elimina el conjunto de semáforos semid junto con su estructura de datos.	Ignorado

9.1.4 Comando IPC_RMID:

Hasta el momento siempre nos hemos referido a un conjunto de semáforos dando su semid. Luego de explicada la system call semctl con el parámetro cmd en IPC_RMID, estamos en condiciones de explicar cual será el tratamiento para los identificadores de semáforos. Cuando un proceso realiza un semget solicitando un conjunto de semáforos nuevo (que no existe todavía en el sistema) deben darse dos condiciones para poder satisfacer el system call:

1. el proceso solicitante tenga un semid disponible (no haya alcanzado el límite del sistema);
2. la tabla de conjuntos de semáforos contenga al menos una entrada libre.

De finalizar satisfactoriamente esta system call, se le devuelve al proceso el primer semid libre. Cada entrada en la tabla de conjuntos de semáforos posee un descriptor que la identifica. Cuando se otorga un nuevo conjunto; se almacena en el semid que se le da al usuario el descriptor de la entrada de la tabla que se asignó, para mantener la relación, pues dos procesos distintos pueden tener el mismo semid pero refiriéndose a posiciones distintas dentro de la tabla de conjuntos de semáforos.

9.1.5 Comandos SETALL y GETALL:

Debe ponerse especial cuidado al utilizar el system call semctl con cmd SETALL y GETALL. Al realizar este system call se pretende setear u obtener TODOS los valores de los semáforos individuales del conjunto referido. Cuando decimos TODOS nos referimos a la cantidad definida de semáforos en el conjunto. Esta cantidad es la especificada al realizar el system call semget que crea este semáforo como uno nuevo. Puede suceder que un proceso realice un semget sobre un semáforo ya existente especificando una cantidad de semáforos individuales menor a la definida en el conjunto. Esto no es un error sino que es la semántica correcta de semget. Sin embargo para realizar un SETALL o GETALL debe operarse sobre TODOS los semáforos del conjunto. Si el llamador define un array de tamaño inferior a la dimensión real del conjunto los resultados del system call son impredecibles, ya que va a recibir un puntero a zona de usuario. De allí va a leer una cantidad de datos que son determinados por la dimensión del conjunto de semáforos. Tal es la especificación hallada en la bibliografía. Se supone que el proceso que realiza un SETALL o GETALL conoce en realidad la dimensión verdadera del conjunto.

10. Anexo C: Llamada a Procedimiento Remoto - Remote Procedure Call

10.1 Introducción

Una enorme proporción de la interacción entre procesos en sistemas distribuidos se realiza a través de operaciones remotas: unproceso envía un mensaje a otro con un pedido y el segundo responde con una respuesta o un acuse de recibo. Aunque en sistemas centralizados este tipo de operaciones pueden encontrarse -si consideramos una llamada al sistema como un pedido de un proceso que es respondido por el sistema operativo- una importante diferencia en un sistema distribuido, es el mecanismo de comunicacón interprocesos.

En caso de un único procesador, se asume la existencia de memoria compartida, hasta la sincronización de dos procesos requiere que haya un área en común donde se va a alojar la variable de semáforo. En un sistema distribuido esto no es posible.

Las operaciones remotas tienen básicamente la misma estructura que las llamadas a procedimientos. La Llamada a Procedimeintos Remotos -RPC-, son una manera sencilla de empaquetar el pasaje de mensajes y hacerlo ver como si fuera parte de la programación convencional, por lo tanto fácil de utilizar.

10.2 Explicación general del mecanismo

El modelo cliente-servidor sufre de una debilidad, que todo su paradigma de comunicaciones está basado en sentencias de entrada/salida (send/receive) que deben ser manejadas por las aplicaciones.

El objetivo es que los procesos puedan hacer llamados a procedimientos que no son locales a la máquina donde estan corriendo. Cuando un proceso A llama a una subrutina que se encuentra en otra máquina, el proceso A se queda bloqueado hasta que no se ejecute este procedimiento, aunque se encuentre en la máquina B -tener en cuenta, que cuando un proceso llama a una subrutina no sigue ejecutando en paralelo, simplemente espera a que esta ejecute y le retorne un resultado-. De esta forma, toda la información necesaria es transportada en los parámetros, así ni el pasaje de mensajes ni instrucciones de I/O son visibles al programador. Esto se llama **Llamada a Procedimiento Remoto, Remote Procedure Call**, o simplemente ***RPC***.

10.2.1 Operación de RPC básica

Para entender mejor RPC, primero se describirá una llamada a un procedimiento local dentro de un programa. La secuencia puede seguirse en la Figura 9.

Ej `count = read (fd, buf, nbytes);`

Para llevar a cabo esta llamada, se hace un push de los parámetros al stack del programa en el mismo orden, es decir primero nbytes, luego buf, y por último fd, así al hacer pop de los parámetros del stack el primero que se saca es fd (Convención de llamada "C"). Una vez que *read* haya ejecutado, coloca el código de retorno en el stack y saca el Return Address, transfiriendo el control al llamador.

En C, por ejemplo, pueden pasarse parámetros por valor y por referencia. Por valor significa que se copia el valor del parámetro en el stack, la subrutina puede modificarlo pero en realidad nunca llega esta modificación al llamador. Por referencia, por el contrario, en vez de enviar el valor se guarda en el stack la dirección de la variable, así cada modificación que se realice dentro del procedimiento a esta variable, será visible en el momento que el control retorne al llamador.

Esta diferencia entre parámetros por valor y por referencia, es de fundamental importancia para RPC. Debe mantenerse la transparencia para rutinas locales y remotas. Para lograr esta transparencia, cuando se compila un programa y se detecta que hay un llamado a un procedimiento remoto, en vez de adicionar código de alguna librería como sería el caso para *read*, se une un tipo de código especial llamado ***client stub*** . Este se va a encargar de empaquetar los parámetros y enviar un mensaje al servidor. Una vez enviado este mensaje, el código del stub ejecuta un receive bloqueándose a sí mismo en espera de una respuesta.

El código del **stub server** ejecuta una instrucción de receive y queda bloqueado en espera de la llegada de un requerimiento. Cuando el mensaje llega al servidor, el kernel se lo pasa al stub del servidor quien desempaqueta los parámetros y llama a la rutina del servidor que va a procesar el requerimiento de forma local, colocando los parámetros en el stack. En el caso del *read*, esta rutina pondrá los datos leídos en un buffer, que forma parte del área de datos del stub del servidor, así cuando termina la ejecución, puede volver a empaquetar los parámetros y el resultado para enviarlos al stub del cliente. Al recibir este mensaje, el stub del cliente se desbloqueará, desempaquetará los resultados y los escribirá en el área donde realmente los espera el cliente. De esta forma, para el cliente es indiferente si se trata de una ejecución local o remota.

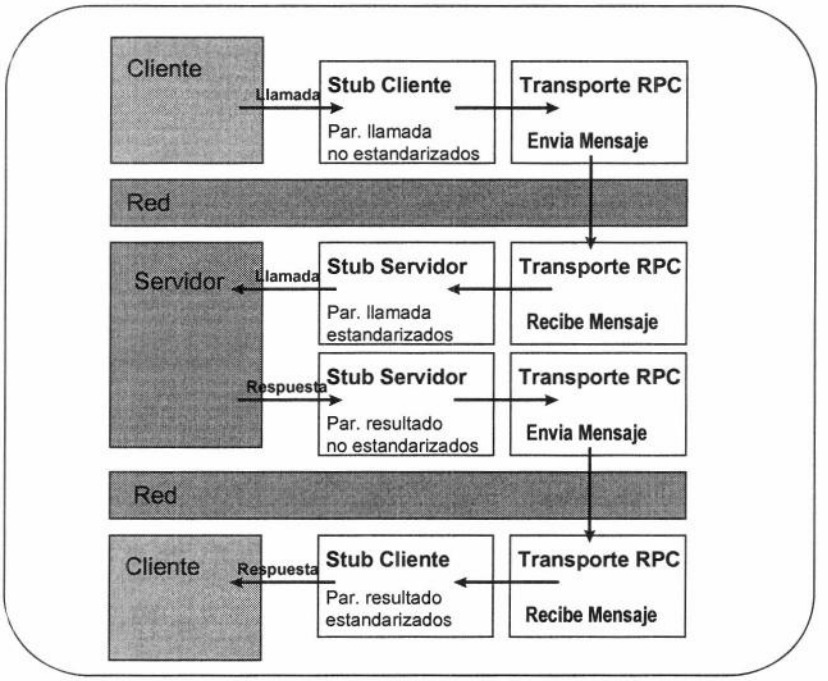


Figura 9: Estructura de una llamada a procedimiento remoto

10.2.1.1 Secuencia de acciones de una llamada de RPC.

La secuencia de acciones generada por una llamada RPC se puede ver en la Figura 9.

1. El procedimiento cliente llama al stub cliente de manera transparente y local, usando el stack.
2. El stub cliente arma el mensaje y se lo envía al kernel.
3. El kernel en el cliente realiza el send del mensaje al kernel de la máquina remota.
4. El kernel remoto le pasa el mensaje al stub del servidor.
5. El stub del servidor desempaqueta los parámetros y se los pasa al servidor, usando el stack.
6. El servidor propiamente dicho realiza su trabajo y retorna un resultado al stub.
7. El stub del servidor empaqueta el valor retornado y se lo manda al kernel.
8. El kernel remoto envía el mensaje al kernel del cliente.
9. El kernel del cliente retorna el mensaje al stub del cliente.
10. El stub cliente desempaqueta el resultado y se lo pasa al cliente.

El paso de los mensajes por los respectivos stubs es transparente tanto para el cliente como para el servidor.

Un mismo servidor puede proveer varias funciones, por lo tanto cada vez que llega un mensaje al stub del servidor, debe chequearse cuál es el servicio que se requiere, siendo esta información parte del mensaje enviado por el stub del cliente.

10.2.2 Pasaje de Parámetros.

Todo funciona de manera bastante sencilla siempre que el cliente y el servidor ejecuten en máquinas idénticas y los parámetros pasados sean de tipo escalar, como ser enteros, caracteres o booleanos.

Esto no es así en la mayoría de los casos, por ejemplo, IBM utiliza EBCDIC como código de caracteres de sus mainframes y ASCII como código para sus PCs. Como consecuencia que no se puede pasar un caracter de una PC a un mainframe directamente. Es necesario algún tipo de mapeo o traducción. También hay problemas con números de punto fijo.

Por este motivo se estableció una forma **canónica** para el pasaje de enteros, booleanos, caracteres, de forma tal que todos los que empaquetan parámetros para poner en un mensaje deben primero hacer esta transformación canónica de todos ellos. De esta forma no es necesario saber si la máquina remota trabaja con EBCDIC o con ASCII ya que hay un formato fijo para todo tipo de parámetros que es independiente del hardware donde se estén ejecutando.

El problema acarreado es la eficiencia. Una solución a este problema podría ser la de añadir a cada mensaje el tipo de máquina que está enviando el mensaje, sin ningún tipo de transformación previa. El receptor, chequea este dato, si coincide con su tipo de máquina, lo pasa al servidor y sigue el camino, sino, hace la conversión a su propio tipo de representación. La desventaja es que cada receptor debería conocer el detalle de implementación de cada proveedor y mantenerse actualizado de posibles cambios.

10.2.2.1 Introducción al uso de XDR

Como las computadoras no usan una representación de datos común, los servidores y sus clientes deben solucionar las cuestiones derivadas de la representación. Para resolver este problema la interacción de los servidores con los clientes puede ser asimétrica o simétrica. La conversión asimétrica requiere que el cliente o el servidor conviertan entre su representación propia y la representación nativa de la otra máquina. La conversión simétrica usa una representación estándar a la red y requiere que tanto el cliente como el servidor, conviertan entre la representación estándar de la red y la local a la máquina.

El problema principal de la interacción asimétrica radica en que se necesitan múltiples versiones de cada programa. Si la red soporta n arquitecturas, la interacción asimétrica requiere un esfuerzo de programación de orden n^2. La interacción simétrica podría causar un poco más de overhead pero provee interoperabilidad con un único programa por arquitectura, requiriendo un esfuerzo de programación proporcional a n.

Sun Microsystems ha definido una representación de datos externa (eXternal Data Representation) que se ha convertido en un estándar de facto conocido como XDR. Provee tanto definiciones para datos no escalares (arreglos y estructuras) como para datos básicos (enteros y cadenas de caracteres). Las librerías de XDR proveen conversión entre

la representación de datos nativa a una máquina y la representación estándar y viceversa. Los clientes y servidores pueden usar las funciones de XDR para convertir los datos a representación estándar antes de enviarlos y a su formato nativo una vez recibidos. La mayoría de los programadores podrían escribir sus propias rutinas de conversión. Dado que algunas requieren de bastante experticia -convertir la representación de punto flotante nativa a una máquina a la de XDR sin perder precisión requiere conocimientos de análisis numérico básico-. Por este motivo se proveen rutinas de librería que efectúan la conversión. La mayoría de las implementaciones de XDR utilizan el paradigma de bufferización que le permite al programador crear un mensaje completo en formato XDR.

10.2.3 Cómo agregar código RPC.

Generalmente se hace automáticamente. Se define un archivo donde se especifica el nombre y los tipos de parámetros de las funciones remotas al cliente, como el número de programa (o servidor). Esto se logra teniendo un compilador que lee las especificaciones y genera tanto el stub cliente como el stub del servidor. El código generado empaqueta y desempaqueta, según corresponda, los parámetros en un formato de mensaje oficialmente aceptado. Teniendo el código generado para ambos stubs a partir de una misma especificación, simplifica la programación y reduce las probabilidades de error haciendo transparente al sistemas las dificultades subyacentes relacionadas al pasaje de parámetros.

Finalmente, el último problema es resolver cómo pasar punteros. Hay que tener en cuenta que un puntero sólo tiene sentido en el espacio de memoria donde se encuentra el proceso, fuera de ahí no significa nada. La dirección x1000 del cliente no contiene los mismos datos que la dirección x1000 en el servidor.

Una solución es impedir el pasaje de punteros. Pero son tan usados que es casi imposible obviarlos. Una aproximación puede ser la de pasar los elementos del vector en su totalidad dentro del mensaje, así el stub del servidor podría llamar a la rutina apuntando al lugar propio donde guardó el vector que le pasaron en el parámetro siendo esta dirección diferente a la que se tenía en la máquina del cliente. El servidor a su vez escribe sobre esta área de memoria y el stub servidor se encarga de devolverlo al cliente con la copia modificada. De esta forma un llamador por referencia se suplanta por un **copy/restore**.

Existe una optimización que lo hace más eficiente. Si el stub sabe si el buffer es un parámetro de entrada o de salida, una de las copias puede ser eliminada. Es decir, si es un parámetro de entrada, no hay que hacer la copia de vuelta, caso contrario, no hay que hacer la copia en la llamada. Esto se define en el lenguaje de especificación formal cuando se definen las funciones remotas y sus parámetros.

10.2.3.1 Utilización de RPCGen para la generación de programas con RPC.

El modelo de llamada a procedimiento remota hace a los programas distribuidos más fáciles de diseñar y comprender porque relaciona la comunicación cliente-servidor a

una llamada a procedimiento convencional. El RPC ve a cada servidor implementando uno o más procedimientos. Un mensaje enviado desde un cliente a un servidor corresponde a una llamada a un procedimiento.

Como los procedimientos convencionales, los remotos aceptan argumentos y retornan uno o más resultados. Los argumentos y resultados intercambiados entre el llamado y el llamador proveen una definición precisa de la comunicación entre el cliente y el servidor.

Usar este modelo ayuda a los programadores a enfocar sus esfuerzos en la aplicación en lugar de en el protocolo de comunicación. El programador puede construir y probar un programa convencional para resolver un problema y luego dividirlo en partes que ejecuten en diferentes máquinas. Esto constituye una visión muy simplificada de lo que significa diseñar y construir una aplicación cliente-servidor ya que la transformación para utilizar RPC no es tan directa ni sencilla. El programador puede elegir seguir la especificación haciendo código desde el comienzo, usar procedimientos almacenados en librerías de RPC o bien utilizar una herramienta que genera automáticamente los programas denominada rpcgen.

RPC permite construir un programa convencional y luego transformarlo en un programa distribuido por medio de llevar algunos de los procedimientos a una máquina remota. Se pueden minimizar los cambios y reducir la posibilidad de errores incorporando procedimientos stub a los programas. Estos implementan la comunicación necesaria permitiendo que los procedimientos originales se mantengan invariables.

Debido a que la mayoría de los programas que utilizan RPC mantienen el mismo diseño, rpcgen puede generar gran parte del código requerido en forma automática. Además de crear un archivo de especificación, sólo es necesario proveer algunos procedimientos de interface y los de la aplicación en sí. Rpcgen genera los procedimientos relacionados a RPC como ser registrar al servidor con el port mapper, proveer comunicación entre cliente y servidor y enviar las llamadas entrantes al servidor correspondiente.

10.2.4 Binding Dinámico

Falta mencionar cómo hace el cliente para localizar al servidor. Existe una forma simple consistente en fijar la dirección del servidor en el código del cliente. Esto es extremadamente inflexible. Cualquier modificación implica que se tenga que recompilar el código del cliente para poder volver a conocer el lugar donde se encuentra su servidor.

Para solucionarlo se utiliza el llamado **binding dinámico**. Lo primero que hay que hacer para poder implementar este tipo de binding, es declarar en la especificación formal el nombre del servidor, el número de versión, y una lista de procedimientos provistos por este servidor. Para cada procedimiento, los tipos de los parámetros que utilizan, y si son de entrada, salida o entrada/salida.

Cuando el servidor comienza su ejecución, efectúa una llamada a una rutina propia de inicialización, fuera del ciclo principal, que se encarga de realizar un **export** de la interfaz del servidor. El servidor envía un mensaje a un programa llamado **binder**, para

indicarle que lo de de alta y que conozca de su existencia, *registrándose* al binder. Para ser registrado el servidor debe proveer los siguientes datos: el nombre del servidor, su número de versión, un identificador único -de 32 bits de longitud- y un manejador que va ser utilizado para localizarlo. Este manejador es dependiente del sistema, y puede que sea una dirección Ethernet, una dirección IP, u otro valor. También en este momento se proporciona información para la autentificación de los clientes en caso de ser utilizada. Existe, también, la operación inversa: desregistrarse.

Cuando el cliente hace una llamada a una función remota por primera vez, el stub cliente se da cuenta que todavía no tiene hecho el bind al servidor, entonces manda un mensaje al binder pidiendo que haga un **import** de una versión determinada de la interfaz del servidor cuyo nombre especifica. El binder revisa toda su tabla de servidores que hayan exportado una interfaz que coincida con lo que el cliente está buscando. Si no existe ningún servidor con esas características, la llamada al procedimiento retorna un error. En caso contrario, el binder retorna un manejador y un identificador único al stub del cliente. El stub utiliza el manejador como dirección a dónde enviar el mensaje. El mensaje contendrá los parámetros y el identificador único que le devolvió el binder. Este identificador será usado por el kernel para direccionar automáticamente el mensaje al servidor correcto.

Este método de exportar e importar interfaces es muy flexible. Permite asimismo facilidades para aplicar un método de autentificación a cada cliente que quiera hacer un bind con los servidores.

La desventaja principal la presenta el overhead extra para exportar e importar las interfaces. Es significante si los clientes tienen poco tiempo de ejecución, pues esto influye en forma significativa en su performance.

10.2.5 Semántica de RPC en presencia de fallas

RPC logra su objetivo en lo que respecta a hacer transparente la diferencia de una llamada a un procedimiento local o remoto dentro de un programa. Sin embargo, ante eventuales fallas, los problemas son difíciles de enmascarar. Es decir, a diferencia de llamadas locales, lo procedimientos remotos tienen asociados una serie de inconvenientes extras. Puede suceder que:

1. El cliente no puede localizar al servidor.
2. Se pierde el mensaje de requerimiento del cliente al servidor.
3. El mensaje de respuesta del servidor se pierde.
4. El servidor se cae luego de recibir el requerimiento o de enviar la respuesta.
5. El cliente se cae.

1- Localización del servidor

El mayor problema luego de identificar el error al tratar de hacer el bind con el servidor, es el de realizar una rutina que maneje este error. Esto rompe con el modelo de transparencia que brindaba el RPC, ya que el programador tiene que codificar qué hacer en caso de tener un error en el bind con el servidor. Esto provoca que todo lo mencionado referido a que los procedimientos remotos vía RPC se tratan igual que los locales pierda sentido.

2- Se pierde el pedido al servidor

El kernel pone un timer cuando envía el mensaje. Si no recibe una respuesta o un acknowledge dentro del lapso definido, se retransmite el mensaje. Si realmente se perdió el mensaje, el servidor toma la retransmisión del requerimiento como la original.

3- Se pierde la respuesta al cliente

Se vuelve a depender del timer, aunque esta vez sea más difícil. El cliente vuelve a enviar el requerimiento. El problema está en que el kernel del cliente realmente no sabe si no hubo respuesta, si se perdió el pedido, la respuesta, si el servidor está lento, etc.

Hay algunas operaciones que se pueden repetir varias veces sin que ocasione ningún daño. Se las llama **idempotentes.** Un ejemplo consiste en sumar dos números y devolver el resultado: se puede ejecutar tantas veces como sea necesario sin traer problemas y siempre con idéntico resultado.

Consideremos, sin embargo, el caso de una sucursal bancaria pidiendo al servidor que transfiera un millón de dólares de una cuenta a otra. Si el requerimiento llega al servidor y éste lo ejecuta, y se pierde la respuesta, el cliente nunca se va a enterar que la transacción fue realizada, y va a retransmitir el mensaje, ocasionando que se transfieran dos millones en vez de uno.

Una posible solución es que el kernel ponga un número de secuencia en el mensaje pudiéndose identificar cuál es original y cuál copia y manejar si ya se ha procesado o no el pedido. Otra alternativa consiste en ponerle a cada mensaje un header, que identifique si es el original o es una copia.

4- Falla del servidor

Este problema también se relaciona con la propiedad de idempotencia, aunque no puede solucionarse simplemente con números de secuencia. Consideremos el caso en que el requerimiento llegue al servidor, éste lo ejecute y falle antes de enviar la respuesta. O falle justo antes de llevar a cabo el pedido.

Existen tres formas de atacar este problema:

- Esperar a que el servidor se recupere, y mandar la operación otra vez. Se sigue intentando hasta que se recibe una respuesta. Esta técnica se llama **at least once** semantics -al menos una vez- y garantiza que el RPC se haya

Las técnicas que involucran la elimiación de los huérfanos no son sencillas y traen problemas adicionales ya que los procesos pueden tener recursos reservados, llamadas de RPC establecidas hacia otros procedimientos, etc.

11. Bibliografía

11.1 Material Referenciado por Orden de Aparición

[012] [DEA/95] Dean T.R. and Cordy J.R. "A Syntactic Theory of Software Architecture." In *IEEE Transactions on Software Engineering, Vol 21, No 4*, pages 302-313. Abril 1995.

[028] [GAR/93] Garlan D. and Shaw M. "An Introduction to Software Architecture" *Advances in Software Engineering and Knowledge Engineering.* New York: World Scientific, Vol. 1 pp. 1-39,1993.

[013] [GAR/95] Garlan D., Allen R. and Ockerbloom J. *Architectural Mismatch or Why it's hard to build systems out of existing parts*, Computer Science Department, Carnegie Mellon University, 1995.

[029] [SHA/89] Shaw M. "Larger Scale systems require higher-level abstractions", *Proc. Fifth Int. Workshop on Software Specification and Design*, IEEE Computer Society, pp 143-146, 1989.

[011] [KRA/94] Kramer J. *Distributed Software Engineering: Invited State-of-the-Art Report*, Department of Computing, Imperial College, London, 1994. (IEEE).

[030] [BLA/95] Blakeley B., Harris H. and Lewis R.*Messaging and Queuing Using the MQI.*McGraw-Hill Series on Computer Communications, 1995.

[010] [SIL/xx] Silberschatz, Peterson, Galvin. *Operating Systems Concepts. Third Edition*. Addison Wesley.

[001] [TEL/94] Tel G. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[003] [SIN/94] Singhal M., Shivaratri N. *Advanced Concepts in Operating Systems*. McGraw Hill. 1994.

[027] [DAN/xx] Daniels D., Haskin R. Reinke J., Sawdon W. *Shared Logging Services for Fault-Tolerant Distributed Computing.* IBM Almaden Research Center. Position Paper for Fourth ACM SIGOPS European Workshop.

[025] [WAN/92] Wang Y. and Fucks K. Optimistic Message Logging fot Independent Checkpointing in Message-Passing Systems. In *Proceedings 11th Symposium of Reliable Distributed Systems*, Houston, Texas. Session 6, Octuber 5-7, 1992.

[022] [RUF/92] Ruffin M. KITLOG: a Generic Logging Service. In *Proceedings 11th Symposium of Reliable Distributed Systems*, Houston, Texas. Session 6, pages 139-146, Octubre 5-7, 1992.

[026] [MOU/92] Moura e Silva L and Silva J. G. Global Checkpointing for Distributed Programs. In *Proceedings 11th Symposium of Reliable Distributed Systems*, Houston, Texas. Session 6, October 5-7, 1992.

[002] [ACM/93] *Distributed Systems*, edited by Sape Mullender. ACM Press, Addison Wesley Publishing Company, 1993.

[005] [CHAN/xx] Chandy M., Misra J. *Parallel Program Design - A Foundation*. Addison Wesley.

[006] [TAN/92] Tanenbaum, A. *Modern Operating System*. Prentice Hall. 1992.

[007] [TAN/87] Tanenbaum, A. *Operating Systems Design and Implementation*. Prentice Hall. 1987.

[008] [TAN/91] Tanenbaum, A. *Redes de Ordenadores*. Prentice Hall. 1991.

[009] [COM/87] Comer, D. *Internetworking with TCP/IP Vol III*. 1987.

[031] [COM/91] *Communications of the ACM*, Volumen 34, Número 2. Febrero 1991.

11.2 Material Consultado

[001] [TEL/94] Tel G. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[002] [ACM/93] *Distributed Systems*, edited by Sape Mullender. ACM Press, Addison Wesley Publishing Company, 1993.

[003] [SIN/94] Singhal M., Shivaratri N. *Advanced Concepts in Operating Systems*. McGraw Hill. 1994.

[004] [AND/xx] Andrews, G. *Concurrent Programming - Principles & Practice*. The Benjamin and Cummings Publishing Company.

[005] [CHAN/xx] Chandy M., Misra J. *Parallel Program Design - A Foundation*. Addison Wesley.

[006] [TAN/92] Tanenbaum, A. *Modern Operating System*. Prentice Hall. 1992.

[020] [KAZ/94] Kazman R., Bass L., Abowd G. and Webb M. *SAAM: A Method for Analyzing the Properties of Software Architectures.* Department of Computer Science, University of Waterloo, Software Engineering Institute, Carnegie Mellon University, Texas Instrument INC, 1994.

[021] [GOL/91] Goldberg A. P., Gopal A., Lowry A., Strom R. *Restoring Consistent Global States of Dsitributed Computations*. Distributed Systems Software Technology Group, IBM T.J. Watson Research Center, 1991.

[022] [RUF/92] Ruffin M. KITLOG: a Generic Logging Service. In *Proceedings 11th Symposium of Reliable Distributed Systems*, Houston, Texas. Session 6, pages 139-146, Octubre 5-7, 1992.

[023] [BAC/90] Bacon D. F.*Transparent Recovery in Distributed Systems: position paper*. Abril 1990.

[024] [WOS/xx] Session Summary: *High-Availability Features*. Workshop on Operating Systems in Computer Networks.

[025] [WAN/92] Wang Y. and Fucks K. Optimistic Message Logging fot Independent Checkpointing in Message-Passing Systems. In *Proceedings 11th Symposium of Reliable Distributed Systems*, Houston, Texas. Session 6, pages 147-154Octubre 5-7, 1992.

[026] [MOU/92] Moura e Silva L and Silva J. G. Global Checkpointing for Distributed Programs. In *Proceedings 11th Symposium of Reliable Distributed Systems*, Houston, Texas. Session 6, Octubre 5-7, 1992.

[027] [DAN/xx] Daniels D., Haskin R. Reinke J., Sawdon W. *Shared Logging Services for Fault-Tolerant Distributed Computing.* IBM Almaden Research Center. Position Paper for Fourth ACM SIGOPS European Workshop.

[028] [GAR/93] Garlan D. and Shaw M. "An Introduction to Software Architecture" *Advances in Software Engineering and Knowledge Engineering.* New York: World Scientific, Vol. 1 pp. 1-39,1993.

[029] [SHA/89] Shaw M. "Larger Scale systems require higher-level abstractions", *Proc. Fifth Int. Workshop on Software Specification and Design*, IEEE Computer Society, pp 143-146, 1989.

[030] [BLA/95] Blakeley B., Harris H. and Lewis R.*Messaging and Queuing Using the MQI.*McGraw-Hill Series on Computer Communications, 1995.

[031] [COM/91] *Communications of the ACM*, Volumen 34, Número 2. Febrero 1991.