TESIS DE LICENCIATURA

Modelos Estocásticos para la arquitectura del software de comunicaciones. Implementación de un Framework, y aplicación al estudio de un caso real.

Alejandro Tolomei
Departamento de Computación - Facultad de Cs. Exactas y Naturales
Universidad de Buenos Aires

Director: Dr. Pablo Jacovkis
Departamento de Computación - Facultad de Cs. Exactas y Naturales
Universidad de Buenos Aires

Diciembre, 1999

Abstract

Los procesos estocásticos son una rama de la teoría de las probabilidades que actualmente están siendo cada vez más usados en diversas disciplinas. Originalmente empleados en procesamiento de señales, hoy en día son aplicados en inteligencia artificial, comunicaciones, teoría de la información, computación teórica, diseño de hardware, análisis de algoritmos, y muchos otros campos.

En este trabajo se propone un método basado en sistemas de colas y cadenas de Markov para el diseño de la arquitectura del software de comunicaciones.

Consta de dos partes:

En la primera parte se describe MathSqueak, un Framework desarrollado para estudiar sistemas estocásticos. Se trata de una suerte de laboratorio hecho en Smalltalk, que utiliza y amplía el trabajo del grupo de investigación *Objetos Matemáticos en Smalltalk* de la Facultad de Ciencias Exactas de la Universidad de Buenos Aires.

En la segunda parte se propone el modelo teórico general basado en modelos de Markov para la arquitectura del software de comunicaciones, y luego se aplica a un caso real. Si bien es usual utilizar este tipo de modelos para analizar sistemas de comunicaciones, ésta es una aplicación novedosa pues se modela la arquitectura interna del software.

Cotas de performance y tolerancia a fallas son obtenidas mediante la resolución del sistema con MathSqueak.

Por último, se implementa el software bajo Unix, y tomando mediciones experimentales se validan las predicciones teóricas.

Agradecimientos

A Luciano Notarfrancesco por la valiosa ayuda que me dió en el diseño y el desarrollo de MathSqueak.

A Leandro Caniglia y su grupo de Investigación sobre Objetos Matemáticos en Smalltalk; casi toda la base Matemática sobre la que se apoya MathSqueak proviene de su trabajo.

A Jorge del Teglia, Joaquín Rapela y Juan José Comellas, cuyas incontables ideas, sugerencias y comentarios enriquecieron notablemente el resultado final.

Y finalmente a Pablo Jacovkis, mi director, pues gracias a su claridad de ideas, su profundo conocimiento y su vocación de compartirlo a lo largo de estos años es que este trabajo fue posible.

Contenidos

Introducción

En esta tesis se propone un método basado en sistemas de colas y cadenas de Markov para el estudio de la arquitectura del software de comunicaciones.

En la primera parte se describe MathSqueak, un software que implementa objetos matemáticos para aplicar al estudio de sistemas estocásticos. Se trata de un sistema abierto desarrollado en Squeak, un proyecto de investigación en Smalltalk de Walt Disney Imagineering. En este documento se trata principalmente el diseño y la relación entre los componentes del sistema, sólo se incluyen aquellas partes de la implementación que se consideró especialmente interesantes.

En la segunda parte se propone el modelo teórico general y se estudia un caso real de un sistema de comunicaciones complejo que debe operar en tiempo real. Con un modelo de Markov se analiza el sistema, a fin predecir cotas de rendimiento y garantizar tolerancia a fallas.

Luego aplicando las herramientas matemáticas provistas por el Framework, se resuelve el modelo teórico aproximado.

Se describe brevemente la implementación dado que ofrece la posibilidad de mostrar un caso práctico de un sistema multithread con sincronización, concurrencia y comunicación entre agentes asincrónicos.

Por último, se validan las predicciones con los resultados experimentales tomados del sistema en funcionamiento.

Objetivos y resultados

Algunos de los puntos a destacar del trabajo:

- La mayoría de los algoritmos númericos implementados provienen de referencias clásicas (como por ejemplo [BURD86], [PRES92]). El valor de este Framework está en la interrelación de los componentes, y en la capacidad de generar herramientas complejas agregando pequeñas contribuciones en forma gradual. Es interesante ver como a partir de los ladrillos elementales del kernel matemático (número, función, intervalo real, etc.) se evoluciona hacia objetos mas complejos tales como polinomios, matrices, ecuaciones lineales, series, operadores de integración, raíces de sistemas no lineales, y otros, hasta llegar finalmente a conceptos tales como cadenas de Markov, procesos autorregresivos, estimadores de intervalos de confianza, etcétera.

- Si bien es común estudiar sistemas de redes de comunicaciones con modelos de Markov, en este trabajo se propone una aplicación nueva de la teoría: el mismo software de comunicaciones es modelado, considerando los componentes internos, su arquitectura y comunicación. Esto es resultado del cuidadoso diseño y de la posibilidad -bastante reciente- de realizar software con agentes asincrónicos independientes corriendo en un mismo programa (programación multithread).

- La implementación permite mostrar un caso real de comunicación, concurrencia y multithreading en un sistema de tiempo real con requerimientos de rendimiento y tolerancia a fallas.

- La aplicación del método científico para validar o refutar con experimetos reales las predicciones de un modelo teórico.

PARTE I

MathSqueak.

Un Framework para procesos estocásticos.

Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.
JOHN VON NEUMAN (1951)

Adde parvum parvo magnus acervus erit
(Agrega poquito a poquito y tendrás una gran pila)
OVIDIO.

Capítulo 1

Diseño del Framework

1.1 Procesos Estocásticos

En esta sección sólo se refrescan algunas definiciones de la teoría de los procesos estocásticos, puede ser salteado sin que afecte la compresión de la tesis.

El material de esta sección proviene de [PAP95] y [SHAN90]. Para un estudio completo de la teoría de los procesos estocásticos pueden consultarse dichas referencias.

Definición: Proceso Estocástico

Es una familia de variables aleatorias $X(t)$, indexadas por un parámetro t, donde t pertenece a un conjunto índice $\mathcal{T}$, que usualmente representa al tiempo. Los valores que toma $X(t)$ son denominados estados y el conjunto de valores posibles se denomina espacio de estados del proceso.

Los procesos estocásticos se pueden clasificar con respecto al espacio de estados: el espacio de estados es discreto, o el espacio de estados es continuo. También pueden ser clasificados según el conjunto índice $\mathcal{T}$, que también puede ser discreto o continuo. Usando las cuatro combinaciones resultantes se pueden representar diversas situaciones del mundo real.

Algunos de los procesos estocásticos más comunes:

- Procesos de Poisson

  Los procesos de Poisson son procesos de parámetro continuo, con espacio de estados discreto. Un proceso de Poisson cuenta el

número $N(t)$ de eventos que ocurren durante el intervalo de tiempo $(0, t]$. Los intervalos entre eventos sucesivos son variables aleatorias independientes y están distribuidos con una misma distribución exponencial.

- Çadenas de Markov

  Se trata de procesos estocásticos con espacio de estados discreto que poseen la propiedad denominada de Markov. Dependiendo del tipo de parámetro se las denomina cadena de Markov (CdM) de tiempo discreto o CdM de tiempo continuo. Un proceso tiene la propiedad de Markov cuando sabiendo el presente, el futuro no depende del pasado, es decir, $\text{Prob}(X(t) = x \mid X(t_0) = x_0 \cap X(t_1) = x_1 \cap \cdots \cap X(t_n) = x_n) = \text{Prob}(X(t) = x \mid X(t_n) = x_n)$ con $t_0 < t_1 < \cdots < t_n < t$.

  Utilizando suficientes estados se pueden aproximar los más diversos comportamientos, pero cada vez es más caro resolverlas analiticamente. En este momento es muy dificil o imposible analizar numericamente cadenas con millones de estados exceptuando casos particulares. Si es posible que millones de estados parezcan un gran cantidad, el número de estados de un modelo markoviano suele crecer exponencialmente con la complejidad del sistema y este tamaño de modelos no es inusual. El camino standard para resolver problemas grandes es la simulación.

  Para especificar una cadena en tiempo discreto es necesario indicar en que estado empieza y cuales son las probabilidades de pasar de un estado a otro. El estado de inicio se suele presentar en forma probabilistica, como una variable aleatoria discreta.

  Las cadenas de Markov cumplen la propiedad de falta de memoria:

$$\text{Prob}(T > t + \tau \mid T > t) = \text{Prob}(T > \tau),$$

  donde T es el tiempo de permanencia en el estado. Por esa razón, en las CdM de tiempo continuo el dicho tiempo de permanencia no tiene memoria y por lo tanto está dado por la distribución exponencial. Para cadenas de Markov de tiempo discreto se llega a la misma conclusión con la distribución geométrica.

  Una cadena suele ser representada como un grafo dirigido y etiquetado. Los nodos representan los estados posibles y los ejes las transiciones posibles. Una transición de a a b es posible si hay una probabilidad positiva de estar en b justo despues de haber estado en a. En CdM discretas, sobre cada transición se anota la probabilidad

que ocurra ese cambio de estado condicionado a que la cadena se encuentre en el estado de donde sale la flecha.

En cadenas de tiempo continuo, como los cambios de estado pueden producirse en cualquier momento, en los ejes están las tasas o velocidades medias de dichos cambios.

Para una clase grande de cadenas se puede demostrar que sin importar el estado inicial del sistema, pasado cierto tiempo el sistema se comporta estadísticamente de la misma manera en todo momento; ese comportamiento motiva el nombre de etapa estacionaria o de equilibrio. Llamemos $\pi_i(t)$ a la probabilidad de encontrar al sistema en el estado i en el tiempo t.

Estar en el equilibrio significa que $\pi_i(t)$ no cambia a medida que t crece. En los casos donde $\pi_i(t)$ tiene límite, cuando pasó suficiente tiempo y las probabilidades $\pi_i(t)$ se acercaron lo suficiente a sus límites, a efectos prácticos, se llegó al equilibrio.

Dicho límite, cuando existe, se denomina π_i e indica la probabilidad de encontrar al sistema en el estado i en la etapa estacionaria.

De nuevo, a los efectos prácticos, si se mira en qué estado se encuentra el sistema en un tiempo lejano en el futuro, la probabilidad de que esté en el estado i es π_i.

El vector (π_i) es llamado distribución estacionaria de la CdM. Para calcularlo hace falta solamente resolver un sistema de ecuaciones lineales. En modelos chicos no existen inconvenientes, ya que se pueden usar los métodos usuales (directos o iterativos). Sin embargo, al crecer el espacio de estados hay que recurrir a técnicas aproximadas más complejas.

Se puede demostrar que π_i es también la proporción del tiempo que la cadena pasa en el estado i.

- Procesos de Nacimiento y Muerte

  Los procesos de nacimiento y muerte son cadenas de Markov donde solo hay transiciones entre estados adyacentes para un cierto orden entre los estados. Por la forma particular de la cadena se puede derivar una fórmula simple para la probabilidad estacionaria de este tipo de procesos.

- Procesos Fluídos

Mirando una cola de espera de muy lejos, con una escala de tiempo muy pequeña, se puede considerar a cada objeto en la cola como un infinitesimal y pensar que la cola es continua en vez de discreta. De esta manera se aproxima un modelo fluído. El proceso posee un espacio de estados continuo con tiempo continuo también. En cada momento, $X(t)$ indica el nivel de fluído en la fila de espera. El servidor tiene una capacidad de drenaje dada, que hace vaciar la cola a una cierta velocidad.

- Procesos Autorregresivos de orden k

  Son procesos con espacio de estados discreto donde $X(n,t) = \sum_1^k \Theta_i X(n-i,t) + e_i(n)$, con $e_i(n) : Normal(0,1)$ y Θ_i: coeficiente arbitrario con valor entre 0 y 1 (dependiendo del problema).

  Cada punto sufre la influencia de k puntos anteriores.

  Es muy usado en tratamiento de señales en comunicaciones.

1.2 MathSqueak: Objetivos de Diseño

Se trata de un sistema abierto, útil para resolver numéricamente problemas con modelos estocásticos, un suerte de *laboratorio* en software. La filosofía con la que está realizado es completamente opuesta a la de los paquetes de Matemática conocidos (Maple^{TM}, Mathematica^{TM}, etc.). Estos programas operan sobre el input como *caja negra* sin indicar el método seguido ni las aproximaciones realizadas para el cálculo.

En cambio, MathSqueak pretende ofrecer un conjunto variado de herramientas al investigador, dejando a su criterio la elección de la más apropiada. Claramente, el sistema exige al usuario un dominio mayor de la matemática , pues requiere conocimiento no sólo del problema, sino también de la conveniencia de cada método en cada situación. Como contrapartida brinda una flexibilidad, y poder de crecimiento sin límites.

Otra diferencia importante es que todo el código esta ahí, visible y al alcance de todos, para revisarlo, corregirlo o hasta eventualmente ampliarlo.

Por último, su diseño abstracto busca darle flexibilidad y a la vez la posibilidad de crecer y evolucionar con un trabajo mínimo de coordinación.

Por este motivo se eligió que sea un Framework. De [GAM94]:

A framework is a reusable design expressed as a set of abstract classes and

the way their instances collaborate. It is a reusable design for all or part of a software system.

It describes not only how to partition the responsibilities of a system among its components, but also how to think about a problem. It is therefore not only a way to reuse code, but a way to reuse design and analysis information, as well.

Frameworks are difficult to design because they are abstract. Designers must look at many concrete applications to ensure that the abstractions that they are designing make sense.

...there is a great advantage to learning a well-designed framework, and mature frameworks (like some of the user interface frameworks) can provide order of magnitude increases in programmer productivity.

1.2.1 Cómo utilizar el Framework

Para utilizar las herramientas provistas por el Framework el usuario debe:

- Ser capaz utilizar el entorno y programar en Smaltallk.
- Decidir en cada caso que método (o combinación de ellos) puede utilizar. Es responsabilidad del usuario ver que las hipótesis requeridas se cumplen para aplicar un operador.

Además puesto que el código fuente está disponible, cualquiera puede adaptar el sistema a sus propias necesidades y gustos.

1.3 Squeak: El lenguaje de implementación

MathSqueak está desarrollado en Squeak, el proyecto Smalltalk actual de todo el equipo original creador de Smalltalk en los años 70.

1.3.1 ¿ Qué es Smalltalk ?

Smalltalk es un entorno de programación gráfico e interactivo creado durante la década del '70 en el centro de investigaciones de Xerox en Palo Alto, California. Consta de un lenguaje de programación basado en la teoría de objetos y un lenguaje de interacción que funciona como interfase entre el sistema de comunicaciones humano y el de la computadora. En

palabras de A. Goldberg, una de sus creadores:

The goal of the SCG [Software Concepts Group] is to create a powerful information system, one in which the user can store, acces and manipulate information so that the system can grow as the user's ideas grow... SCG's strategy for realizing this vision has been to concentrate on two principal areas of research: a language of description (a programming language) which serves as an interfase between the models in the human mind and those in computing hardware, and a language of interaction (a user interface) which matches the human communication system to that of the computer.

(ver [GOLD89] págs. 5-9)

En el lenguaje de programación Smalltalk se sigue un modelo basado en una metáfora muy simple llevada al extremo: objetos y mensajes intercambiados.

Un objeto representa un componente dentro del sistema, como por ejemplo números, rectángulos, diccionarios, etcétera.

Cada objeto cuenta con una zona privada de memoria más un conjunto de operaciones. En la zona privada de memoria existen variables solamente accesibles internamente por el conjunto de operaciones (métodos) propios del objeto. Estas variables, llamadas privadas, determinan el estado interno del objeto a lo largo del tiempo.

Un mensaje es el requerimiento a un objeto para que lleve a cabo alguna de esas operaciones. El objeto receptor del mensaje es quién determina cómo realizar la operación pedida. El conjunto de mensajes que un objeto sabe responder se denomina *interface* y es la única manera de interactuar con dicho objeto.

Además de los conceptos de objeto y mensaje se definen clase e instancia.

Una clase describe la implementación de un conjunto de objetos que representan un tipo dado de componente del sistema. Todos los objetos individuales descritos por esa clase se denominan instancias.

Para implementar este modelo de objetos, clases y mensajes intercambiados, Smalltalk cuenta con un pequeño conjunto de operaciones llamadas *primitivas* que no están expresadas en el lenguaje Smalltalk. Estas primitvas permiten el acceso al hardware subyacente y a la máquina

virtual del sistema que se ocupa de almacear los objetos, asignar y liberar memoria, acceder al sistema de archivos, etcétera (para una explicación detallada del lenguaje ver [GOLD89])

Gracias a este esquema tan simple y uniforme es posible representar entidades matemáticas de manera sencilla y elegante. Como lo expresara Alan Kay ([KAY93]):

Smalltalk's design -and existence- is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages.

1.3.2 Squeak

Se eligió Squeak porque a diferencia de otras implementaciones de Smalltalk, Squeak está evolucionando permanentente, incorpora muchas ideas originales y atractivas tales como Morphic, la máquina virtual se implementa en Squeak, están los fuentes en el mismo entorno, etcétera. De [ING96]:

Squeak is an open, highly-portable Smalltalk implementation whose virtual machine is written entirely in Smalltalk, making it easy to debug, analyze, and change.

Other noteworthy aspects of Squeak include: a compact object format that typically requires only a single word of overhead per object; a simple yet efficient incremental garbage collector for 32-bit direct pointers; efficient bulk-mutation of objects; extensions of BitBlt to handle color of any depth and anti-aliased image rotation and scaling; and real-time sound and music synthesis written entirely in Smalltalk.

Capítulo 2

Kernel Matemático

2.1 Números

Squeak implementa la metáfora de objetos intercambiando mensajes también en aritmética. La clase abstracta *Number* brinda un protocolo externo uniforme a las subclases: *Float*, *Fraction*, e *Integer* (con sus tres subclases *SmallInteger*, *LargePositiveInteger*, *LargeNegativeInteger*) (ver fig. 2.1).

Los números no pueden cambiar su estado, el único estado posible es su valor. La aritmética entera provee precisión infinita, ya que internamente representa números enteros con una cantidad no acotada de dígitos. Esto, sin embargo, provoca un overhead de computación. La clase *SmallInteger* es una implementación optimizada para enteros pequeños (hasta una *word* de la computadora).

Para el soporte de polinomios (ver 2.4) y números algebraicos se agregaron mensajes a *Number*, y la subclase *AlgebraicNumber* hecha por el grupo de Objetos Matemáticos en Smalltalk.

Se implementaron números complejos (la clase *Complex*), pero fuera de la jerarquía de *Number* debido a que no tienen definida una relación de orden.

También se agregó una clase *Infinity* para representar al infinito, sobrecargando los operadores para reflejar la semántica de este símbolo.

Es interesante destacar que cada tipo de número agregado a esta categoría, automáticamente enriquece todos los operadores y objetos que tratan con números, como por ejemplo matrices o ecuaciones diferenciales.

2.2 Funciones

La clase *Function* incluída es la de del grupo de Objetos Matemáticos en Smalltalk (ver [CANI97]).

Un objeto de la clase *Function* contiene un motor (engine) que le permite evaluarse (en cada x devuelve $f(x)$).

Una especialización es la función que se obtiene de una tabla: *TableFunction.*

Una *TableFunction* es una función que tiene como engine un *Dictionary.*

El objeto clave en la implementación de esta clase es el engine. En general puede ser cualquier cosa que sepa responder al mensaje *value* o *value at:* . Los objetos más usados como engines son los Blocks, que internamente contienen la expresión que dado x resuelve $f(x)$.

Solamente agregando el mensaje *value At:* a cualquier objeto ya existente o propio, el usuario puede generar una *Function.*

Esta funcionalidad resulta muy útil para aprovechar la gran cantidad de operadores que trabajan sobre funciones.

Veamos un ejemplo:

Toda variable aleatoria X tiene una función de distribución definida como: $F(x) = \int_{-\infty}^{x} f(y)dy$. Agregamos el mensaje *valueAt:* a la clase de las variables aleatorias y luego para calcular $F(x)$ generamos un *Integrator* (ver 2.3) con la misma variable aleatoria como función a integrar.

2.3 Operadores de Integración

Integrator es un operador que devuelve la integral de una función calculada mediante un método numérico.

Dada una *Function*, y un intervalo real, un objeto *Integrator* resuelve la integral (númerica) de la función en el intervalo.

Tipos de *Integrator*:

- Trapecio : Regla del trapecio (ver [BURD86] y [PRES92])
- Montecarlo: método de Montecarlo ([PRES92])

En la figura 2.1 se diagrama la jerarquía de clases.

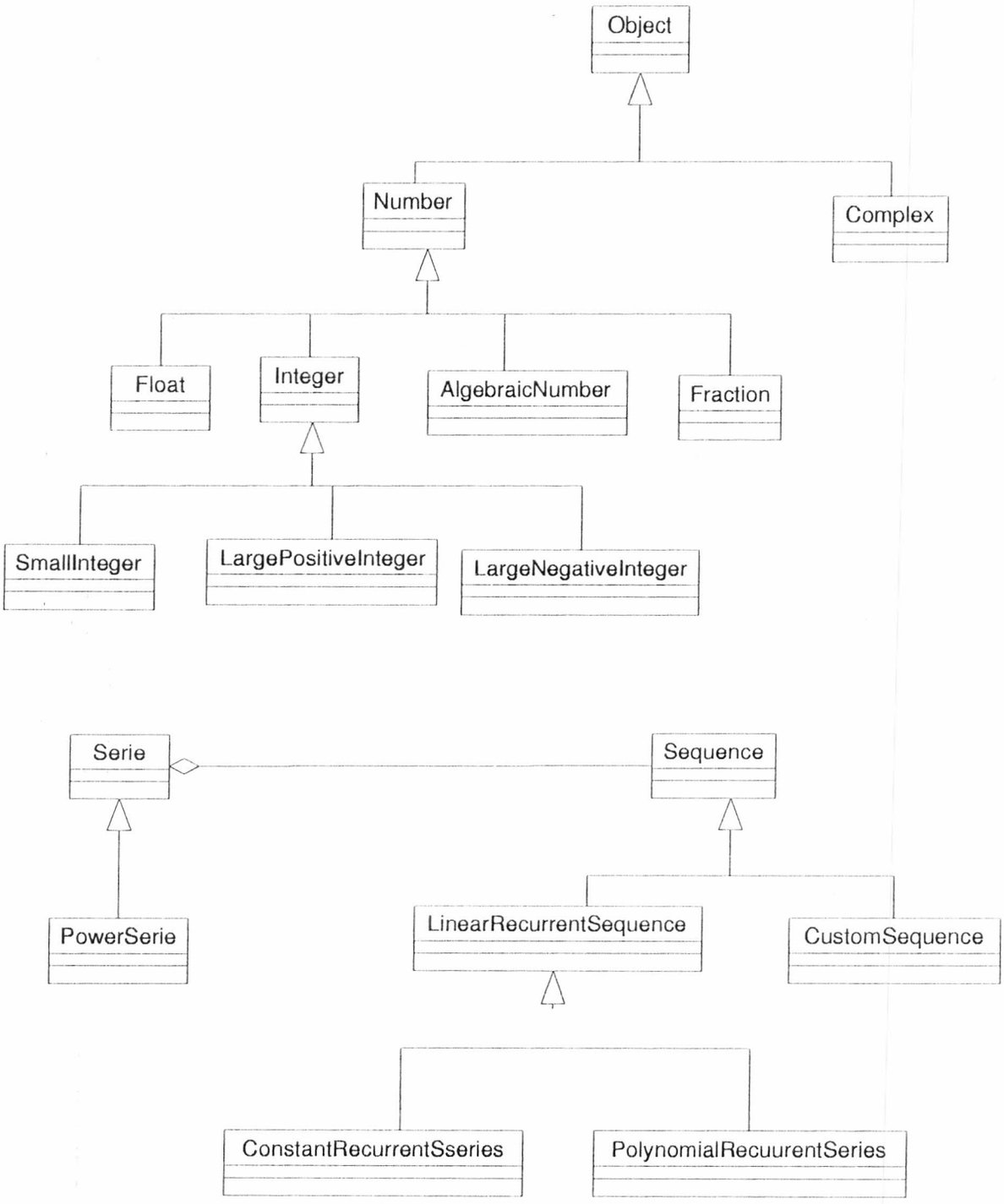
2.4 Polinomios, Sucesiones y Series

Las sucesiones (*Sequence*) pueden ser generadas por extensión, hasta el término n, por recurrencia, o simplemente con un engine que evalúa el término general.

Una Serie (*Serie*) contiene una sucesión, (alguna clase derivada de *Sequence*), puede evaluarse hasta el término n_{esimo}. En particular, se derivó la clase *PowerSerie* para series de potencias.

Los Polinomios (*Polynomial*) pueden ser de una o varias variables. Se soportan todas las operaciones más comunes, y el operador de Sturm-Liouville para hallar ceros de polinomios con coeficientes racionales (ver fig. 2.1).

Figura 2.1: Kernel Matemático



Capítulo 3

Algebra lineal y ecuaciones no lineales

3.1 Algebra lineal

Implementa espacios vectoriales, subespacios, bases, matrices, matrices ralas, ortogonalización de Gram-Schmit y solución a sistemas de ecuaciones lineales por métodos directos e indirectos (ver [MIS93]).

- Métodos directos
  - Factorización de Matriz (LU)
- Métodos indirectos
  - Gauss-Seidel
  - Método de Jacobi

3.2 Solución de Ecuaciones no lineales

Se incluyen cuatro métodos de resolución de ecuaciones no lineales:

- *Raíces de ecuaciones*
  - Bisección
  - Newton-Raphson
  - Regula Falsi

- *Ecuaciones de Punto fijo* ($f(x) = x$)
  - Iteración de Punto Fijo.

Para una descripción de los algoritmos ver las referencias [BURD86] y [PRES92].

Es interesante notar la gran flexibilidad que permite el uso de *Function* definida en 2.2 (ver figura 3.1 y código fuente) .

En general no existe un *RootAproximator* óptimo, sino que se debe analizar en cada caso cuál es más adecuado. El método de la bisección es el más estable, pues converge siempre, sin embargo es el más lento, y además para poder aplicarlo la función debe ser continua y cambiar de signo en el intervalo bajo estudio (ver [PRES92]).

Normalmente se emplean combinaciones de dos o más métodos para hallar una raíz. Por ejemplo, se parte de un intervalo muy grande, que es achicado con bisección, y luego refinado con Newton-Raphson.

Siguiendo el espíritu del diseño, se deja al usuario la elección del objeto (o combinación de ellos) a usar, siendo además su responsabilidad verificar que se cumplan las hipótesis exigidas por cada uno.

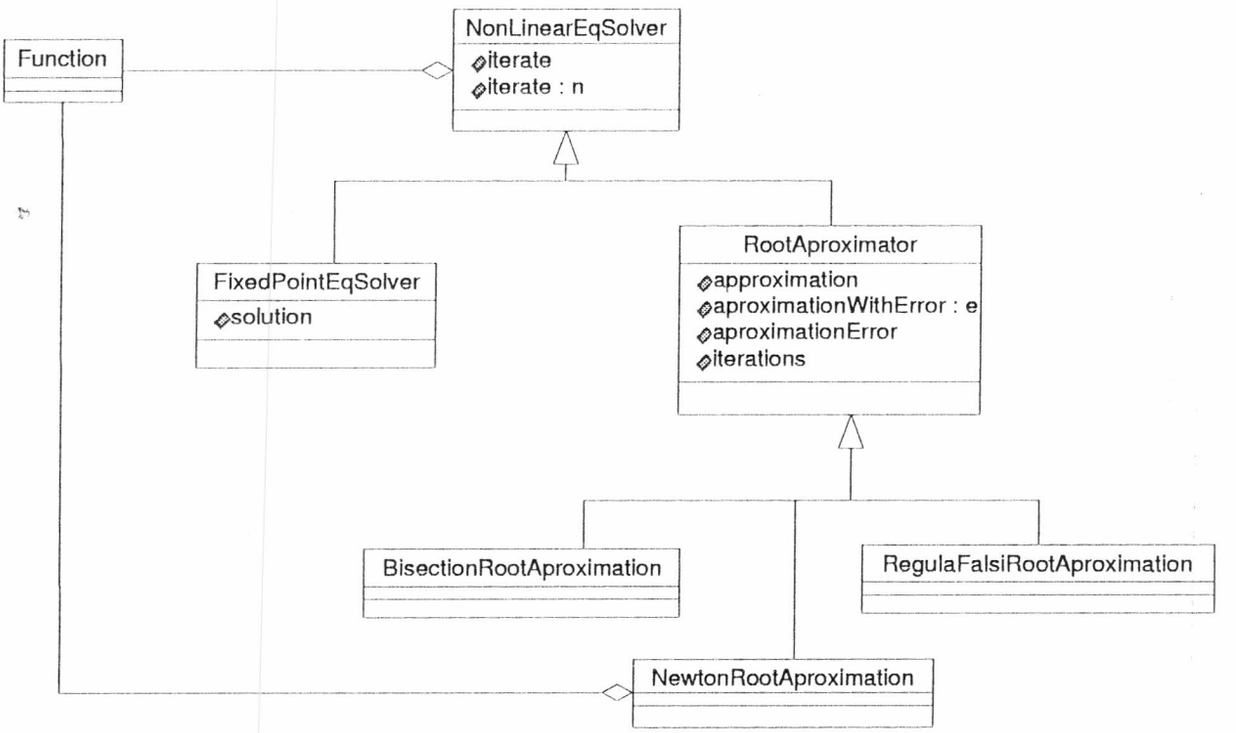


Figura 3.1: Ec. no lineales

Capítulo 4

Probabilidades y Estadística

Hasta este capítulo se describieron los objetos matemáticos generales que sirven al Framework como herramientas para las clases correspondientes a Probabilidades y Estadística.

4.1 Probabilidades

Se implementan las principales variables aleatorias continuas y discretas, para servir en simulaciones y para estadísticos.

La base teórica de la implementación (para variables aleatorias continuas) proviene del siguiente resultado:

Dada una variable aleatoria x con función de distribución $F(x)$ estrictamente creciente, a partir de una variable aleatoria con distribución uniforme U entre 0 y 1 $F^{-1}(U)$ tendrá la distribución requerida (ver [KNUT93]).

Por lo tanto basta con obtener un generador uniforme (U) en el intervalo entre 0 y 1 y luego aplicarle a los valores obtenidos F^{-1}.

Para obtener un generador uniforme se utiliza una generador lineal congruencial (ver [KNUT93]). En nuestro sistema usaremos el que provee el entorno (Park Miller RNG ver [KNUT93]).

Puesto que en general el método de aplicar F^{-1} resulta muy complejo y lento, para casi todas las distribuciones más conocidas se utilizaron

algoritmos generadores más eficientes (tomados de [KNUT93]).

Dada la función de densidad de probabilidad $f(x)$ para obtener la función de distribución $F(x)$ se usa un *Integrator* general; por eficiencia algunas distribuciones sobrecargan la función por una específica.

Distribuciones soportadas:

- Continuas
  - $Gamma(a, b)$
  - $Chi^2(n)$
  - $Erlang(c)$
  - $Exponential(\lambda)$
  - $Normal(\mu, \sigma)$
  - $Uniform(a, b)$
- Discretas
  - $Bernoulli(p)$
  - $Binomial(n, p)$
  - $Geometrica(p)$
  - $Multinomial(v_1 \ldots v_n)$
  - $HiperGeometric(N, M, n)$
  - $Poisson(p)$

todas comparten un protocolo común:

- *density:* x
- *distribution:* x
- *inverseDistribution:* x
- *mean*
- *var*
- *sigma*
- *sample:* n (genera una muestra)

Una aplicación interesante del concepto de polimorfismo de la teoría de objetos puede verse en el mensaje *Distribution: x* .

Por definición $P(X < x)$ es $F(x) = \int_{-\infty}^{x} f(y)dy$. Esto vale para cualquier distribución continua. Lo que varía de una a otra es $f(x)$.

La clase abstracta *ContinousDist* implementa la función (con un operador *Integrator*).

Cada clase derivada sólo deberá redefinir su función de densidad $f(x)$. Al enviarle el mensaje *distribution: x* se ejecutará el código de *Continousdist* que evaluará la $f(x)$ propia del objeto que recibe el mensaje.

4.2 Estadística

Un *estadístico* es cualquier función aplicada sobre una muestra aleatoria X. A partir de la muestra, se puede definir una variable aleatoria $\gamma = g(X)$, que es un estimador puntual del parámetro γ.

MathSqueak incluye un conjunto de estimadores puntuales de media y varianza:

Estimadores de la media:

- con Varianza conocida
  - *NormalWithVarMeanEstimator*
  - *TchebycheffWithVarMeanEstimator*
- con Varianza desconocida
  - *BinomialMeanEstimator*
  - *ExponentialMeanEstimator*
  - *MeanEstimatorWithoutVarStudent*
  - *PoissonMeanEstimator*

Estimadores de la Varianza:

- con media conocida
  - *VarianceEstimatorWithMean*
- con media desconocida
  - *VarianceEstimatorWithoutMean*

Todos ellos responden a los siguientes mensajes:

- *histogram: n*
  Genera un histograma con n subdivisiones entre el valor máximo y mínimo.

- *histogram: n from: a bto: b*
  Es igual sólo que el intervalo considerado es [a,b].

- *mean*
  Media muestral.

- *var*
  Varianza muestral.

- *sigma*
  Desvío standard muestral.

- *confidenceInterval: delta*
  Devuelve el intervalo de confianza de nivel delta.

Según la distribución a la que pertenece la muestra y la información a priori (p. ej. si se sabe de antemano la media o la varianza verdaderas) se deberá usar uno u otro de los operadores soportados (ver ejemplo en 4.2.1).

Unicamente el *TchebycheffWithVarMeanEstimator* -basado en la desigualdad de Tchebycheff- es válido siempre (ver [PAP95]).

El usuario deberá elegir cuál puede utilizar en cada caso.

4.2.1 Ejemplo

Apliquemos dos estimadores para la media de una muestra. Notar que el intervalo de confianza de nivel 0.97 generado por un estimador Normal es más chico que el correspondiente al de Tchebycheff. Esto es esperable ya que el primero es menos restrictivo en cuanto a las hipótesis requeridas; es decir que cuanto más información a priori uno conozca mejor estimación puede obtener.

La respuesta del sistema está en itálicas:

```
n ← Normal standard
s ←n sample: 400
sta ← NormalWithVarMeanEstimator sample: s var: 1.0

sta mean

-0.0478338898845127
sta s2
```

1.07299956801306
sta histogram: 20 from:−2 to: 2

(7 6 11 17 16 19 29 26 30 35 32 30 33 17 21 17 9 10 10 4)
sta confidenceInterval: 0.97

−0.1020433690086566 , 0.1047783110603131 (Xm= 0.0013674710258282O6 err: 0.103410840034484 9)
sta2 ← TchebycheffWithVarMeanEstimator sample: s var: 1.0

sta2 confidenceInterval: 0.97
−0.2873076635689846 , 0.290042605620641 (Xm= 0.0013674710258282O6 err: 0.2886751345948127)

Figura 4.1: Variables Aleatorias

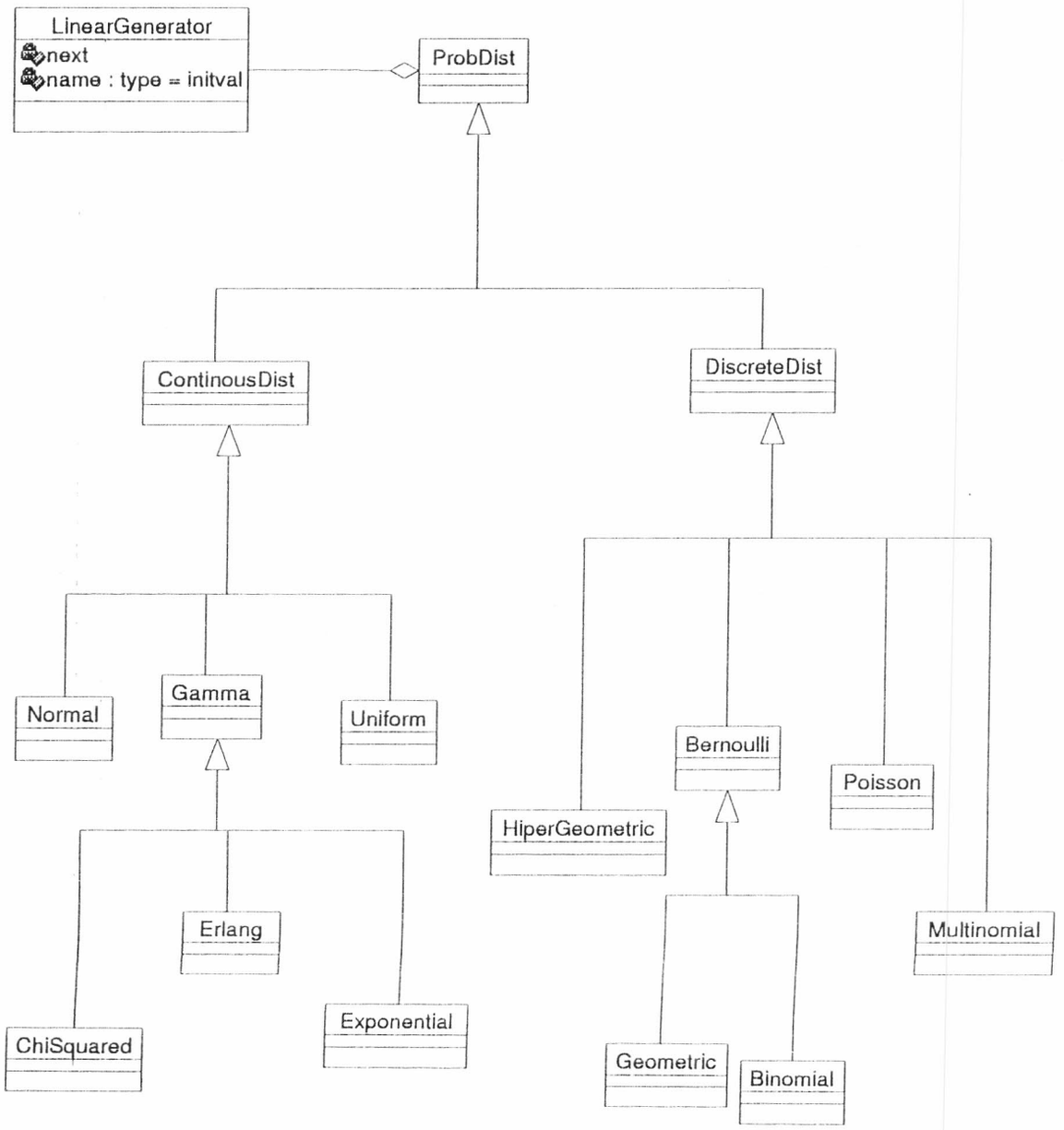
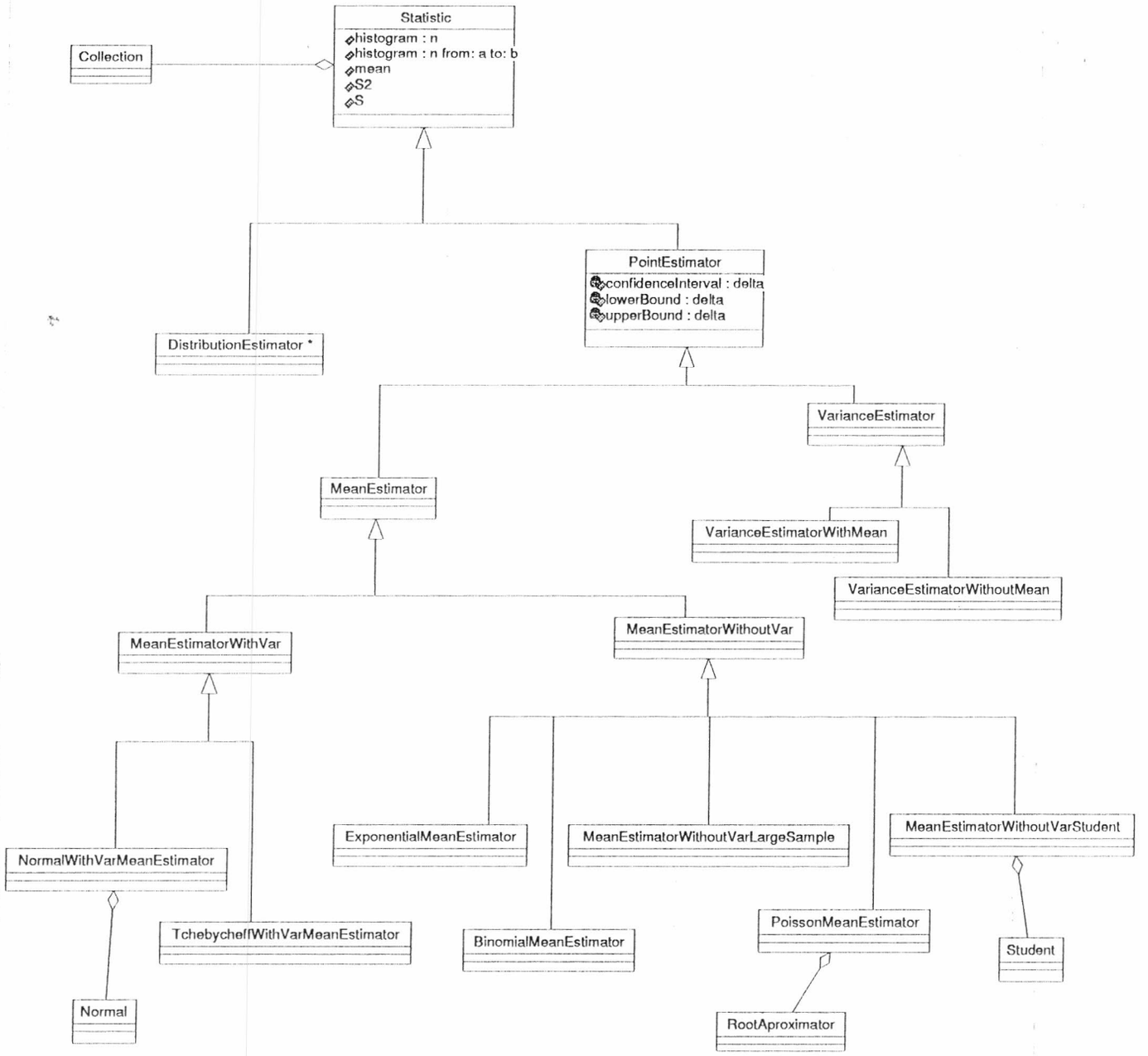


Figura 4.2: Estadísticos



Capítulo 5

Procesos Estocásticos

Todos los procesos estocásticos se derivan de *RandomProcess* (ver fig. 5.1). Este brinda un protocolo básico uniforme correspondiente a las funciones:

- AutoCorrelation(x,y): $R_{xx} = E(X^*(x)X(y))$
- Mean(t): $\mu_x(t) = E(X(t))$
- Sample(n) (genera una muestra de tamaño n)

En la figura 5.1 se muestran todos los procesos incluídos. En particular, se trataron con más detalle las cadenas de Markov discretas y continuas.

Para representar una cadena de Markov el constructor debe recibir la *Matrix* (o *SparseMatrix*) de transición P para el caso discreto o la generadora infinitesimal Q para las continuas (ver [STEW97]).

5.1 Resolución de Cadenas de Markov

Para calcular el vector de probabilidad estacionario de una cadena de Markov irreducible y ergódica se transforma el problema a un sistema de Ecuaciones Lineales (una cadena de Markov es irreducible cuando todo estado puede ser alcanzado desde otro estado cualquiera, y ergódica cuando el número medio de pasos para volver a visitar un estado cualquiera es finito, y además es posible volver a él en un sólo paso justo después de dejarlo. Ver [STEW97]). Los *MarkovChainSolver* resuelven internamente el sistema usando los métodos de Algebra Lineal del capítulo 3.

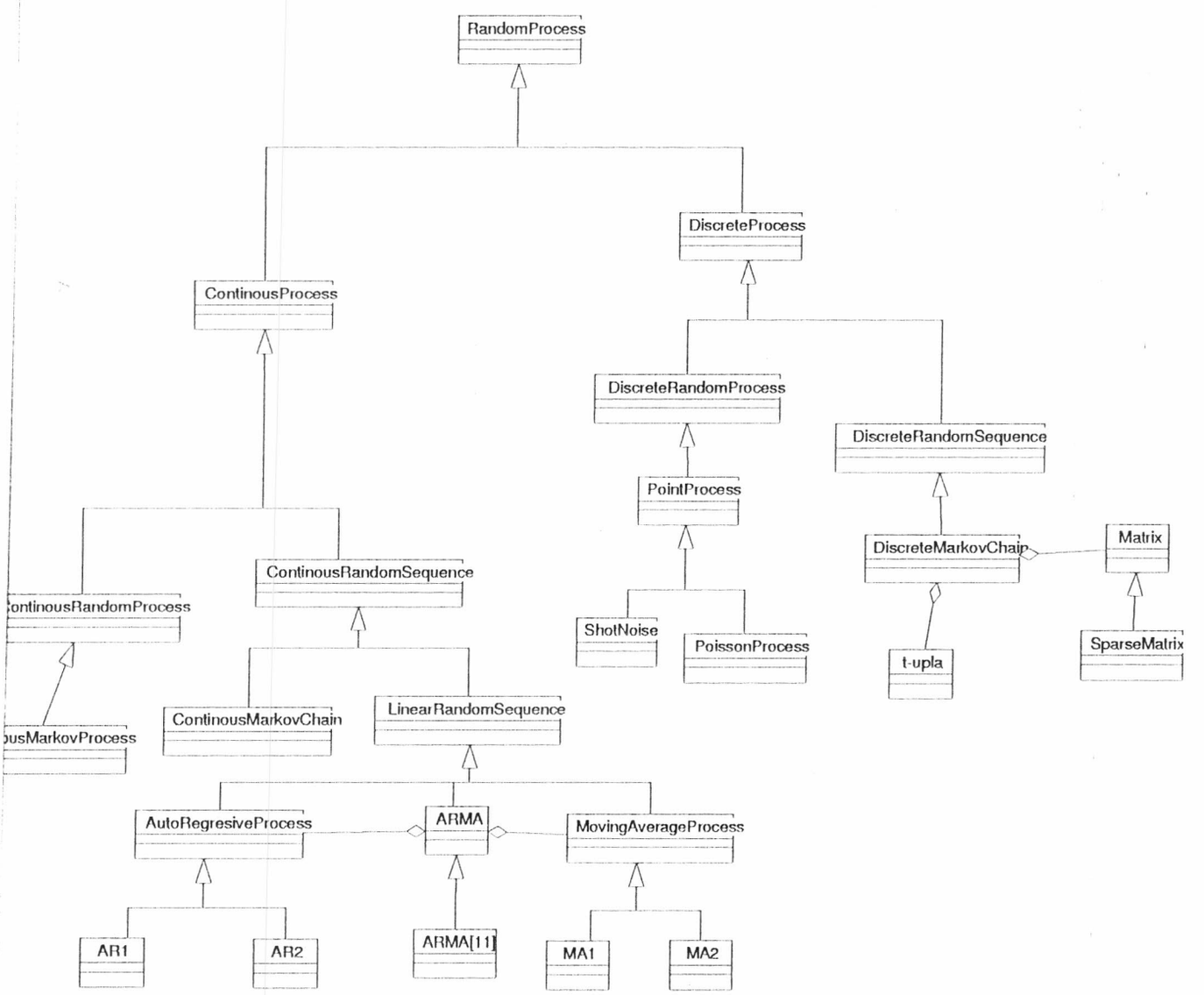
Jerarquía de *MarkovChainSolver*:

- DirectMarkovChainSolver: métodos de resolución directos
  - LUMarkovChainSolver: factorizacón LU (ver [BURD86, STEW97])
- IterativeMarkovChainSolver
  - GaussSeidelMarkovChainSolver: método de Gauss Seidel (ver[BURD86])
  - JacobiMarkovChainSolver: método de Jacobi (ver [BURD86])
  - PowerMarkovChainSolver: método de potencias (ver [STEW97])

Cualquiera de ellos puede ser usado para una CdM dada, que puede contener una *Matrix* o bien *SparseMatrix*.

Se prefirió hacer que los agentes que resuelven una CdM sean externos a la cadena misma para que el usuario tenga la flexibilidad de elegir, probar con varios, y hasta comparar el rendimiento relativo en cada problema.

Figura 5.1: Procesos Estocásticos



Capítulo 6

Simulaciones Discretas

Para simulaciones discretas se diseño un Framework compuesto por tres clases:

- *SimObject*: Objetos activos. Intercambian mensajes. El tiempo sólo avanza al enviar o recibir un mensaje.

- *SimMessage* : Mensajes intercambiados.

- *Simulator* : Coordinador. Contiene todos los objetos de la simulación y avanza el reloj del sistema hasta el próximo evento (cola de eventos).

Para especializar una simulación todos los objetos deben derivarse de SimObject y ser agregados al Simulator. Además los mensajes que intercambian se derivaran de SimMessage.

Dos métodos de las clases base deben ser sobrecargados para agregar la semántica del problema particular:

- *msgIN: aMsg* , El receptor recibe un mensaje de otro objeto
- *msgOUT* , El receptor acaba de enviar un mensaje.

En 6.1 se realiza una simulación de un sistema de comunicaciones formado por un satélite, un filtro receptor (filtro adaptado) y el ruido atmosférico estocástico.

6.1 Ejemplo Simulación Discreta

Se tiene un satelite de comunicaciones que envía pulsos digitales de 1 mseg de duración con ciclos de trabajo de 0.5, y que deben ser detectados en tierra mediante un filtro. La atmósfera introduce dos efectos:

- Actúa como un filtro pasa bajos (Low Pole Filter) con un polo en 10 kHz.
- Agrega ruido aditivo autorregresivo de primer orden con $S/N = 10$ dB.

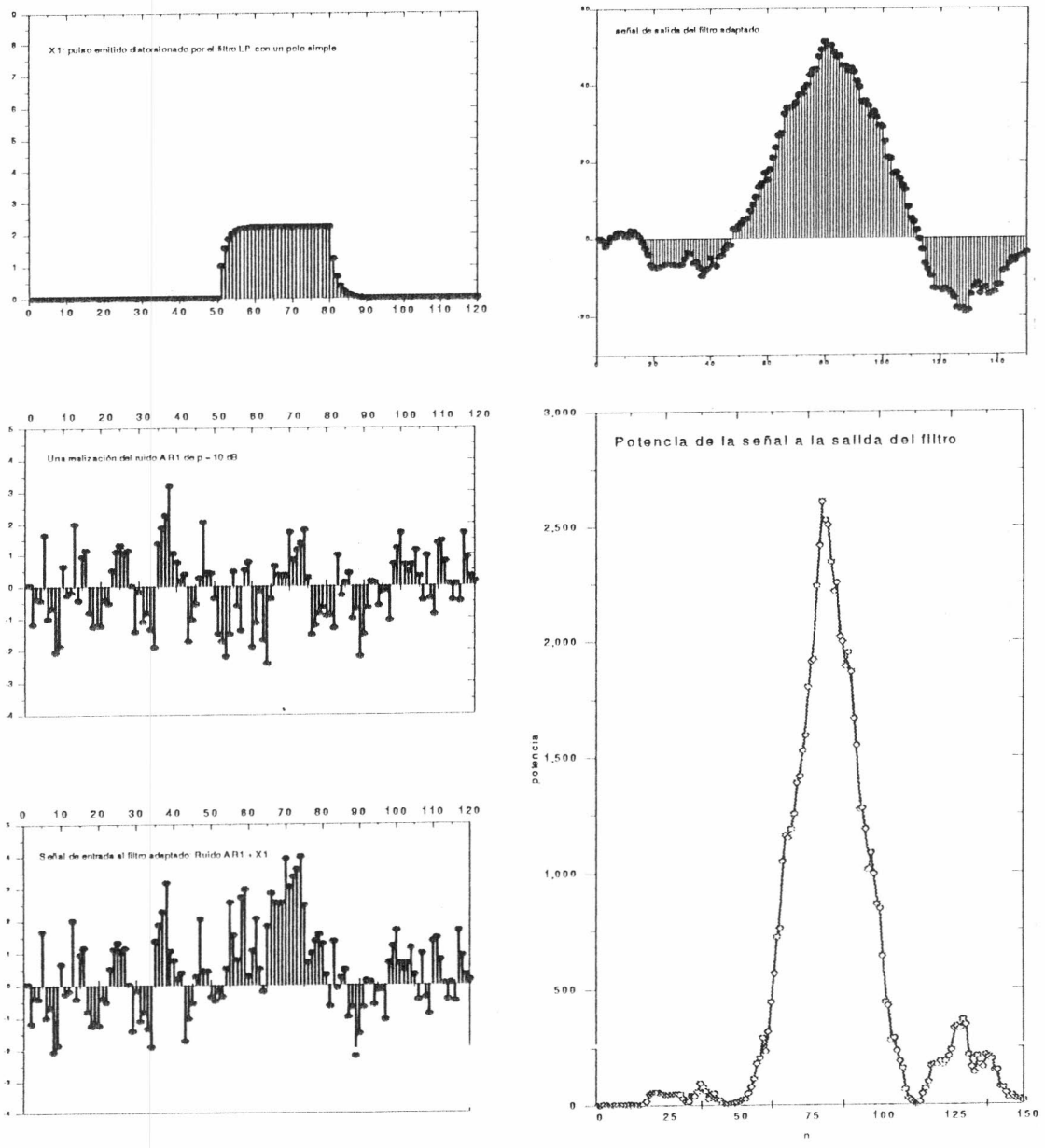
NOTA: Un filtro pasa bajos recibe una señal de entrada S_i y devuelve a la salida $S_o = H * S_i$. H es la transferencia del filtro y su transformada de Laplace es de la forma $L(H)(s) = 1/(s + s_0)$, siendo s_0 el polo.

El receptor detecta la presencia de un pulso a través de un filtro que maximiza la relación potencia de la señal/potencia del ruido en el momento en que todo el pulso se encuentra en el filtro (Matched Filter). La transferencia h será $h = C_n^{-1}Z$ (ver [PAP95]), con C_n: Matriz de covarianza del ruido atmosférico, Z: señal a detectar.

En la figura 6.1 se grafica una realización del experimento con la señal enviada por el satélite, el ruido del medio, y la detección del filtro.

Puede verse en el código fuente del ejemplo la manera de especializar la simulación mediante derivación de las tres clases base.

Figura 6.1: Proceso Estocástico: Simulación de un Filtro Adaptado



PARTE II

Modelos estocásticos para la arquitectura del software de comunicaciones. Aplicación al estudio de rendimiento y tolerancia a fallas de un caso real de un Gateway de comunicaciones

Todos sabemos que el Tetrarca de Galilea posee los mejores zafiros del mundo. Alguien, para robarlos, habrá penetrado por error. Yarmolisky se ha levantado, el ladrón ha tenido que matarlo. ¿ Qué le parece ?
- Posible, pero no interesante - respondió Lonrot. - Usted replicará que la realidad no tiene la menor obligación de ser interesante. Yo le replicaré que la realidad puede prescindir de esa obligación, pero no las hipótesis.

de La muerte y la brújula
JORGE LUIS BORGES.

Capítulo 7

Modelos para la arquitectura del software

En este capítulo se presenta una técnica para modelar la arquitectura del software de comunicaciones usando modelos de Markov. En primer lugar se analiza la utilidad de estos modelos, luego se describen los pasos a seguir para su aplicación y finalmente la manera de resolverlos.

En los siguientes capítulos se muestra una aplicación a un caso real.

7.1 ¿ Por qué es útil un modelo de Markov ?

El software de comunicaciones es un área muy compleja dentro de la informática por las características de los sistemas involucrados:

- Debe operar en tiempo real.
- La implementación y testing son muy complejos por la cantidad de casos posibles.
- Es fuertemente asincrónico pues interactúa con conexiones independientes.
- Con frecuencia debe cumplir requisitos de performance, controles de flujo y de congestión.
- Debe ser flexible para adaptarse a situaciones extremas.

Teniendo en cuenta esto, ¿cómo hacer para garantizar una mínima performance del sistema antes de ser construído ? ¿ Cómo decidir acerca de los recursos mínimos necesarios para soportar no sólo la operatoria normal sino

también situaciones extremas ?

Tomemos el caso de un router del cual deben fabricarse miles de unidades: ¿ cómo determinar la cantidad de memoria necesaria para buffer ? O bien, ¿ cómo elegir entre varias arquitecturas posibles antes de realizar el sistema ?
(En el Apéndice B se describe un caso real de un Servidor de Internet cuya arquitectura se puede determinar a priori a fin de maximizar la performance).

Para este tipo de sistemas se propone un modelo teórico simple basado en cadenas de Markov. Se trata de representar con un proceso estocástico la arquitectura interna del software, su interacción con el hardware y con los nodos remotos.

Se eligió esta herramienta por varias razones. Entre ellas:

- Las cadenas de Markov son simples y poderosas, y cuentan con una base teórica muy fuerte (ver [KLEI75I, KLEI75II])
- Soporta cualquier distribución aleatoria (al precio de incrementar el número de estados) (ver [KLEI75I, KLEI75II]).
- Usando software específico se pueden resolver sistemas con millones de estados.
- El tráfico en redes de comunicaciones se comporta de un modo que resulta fácil de modelar con esta técnica. (ver [HARR93])
- Con las técnicas actuales de programación (multithread y también multiproceso) es posible representar el sistema de manera simple y directa (ver [ROB97]).

En la secciones que siguen se explica el método general para crear un modelo y el mecanismo de resolución del sistema.

7.2 Generando el modelo teórico

Pasos para crear el modelo estocástico:

- Según el diseño elegido para el software determinar los agentes independientes que existirán en el sistema.
- Modelar el mecanismo de comunicación entre ellos (Normalmente mediante una cola de mensajes).
- Modelar el tráfico de llegada y salida en cada extremo con el exterior.
- Definir las distribuciones probabilísticas de cada agente.
- Si es necesario, representar el mecanismo interno de la misma manera, pueden incluirse en el modelo los componentes de hardware (por ejemplo el acceso a disco, carga del procesador etc. ver [AGRW87]).

En general los flujos de datos en comunicaciones son bien conocidos por lo que para una gran cantidad de casos ya se conocen las distribuciones usuales. Igualmente se pueden aplicar las técnicas standard de estimación de distribuciones para el tráfico si resulta necesario (ver [PAP95, HARR93]).

Finalmente queda resolver el modelo para encontar las variables relevantes.

7.3 Resolviendo el sistema

Luego de definido, el sistema puede ser transformado en una cadena de Markov. Las alternativas para resolverlo son:

- Por deducción directa exacta o aproximada (aunque no siempre es posible)
- Generando la cadena de Markov y resolviéndola con programas específicos (en forma exacta o bien aproximada).
- Hallando los parámetros requeridos mediante simulaciones.

En este trabajo presentamos un caso real que puede ser atacado con las tres técnicas.

Capítulo 8

Diseño del gateway

8.1 Introducción

En lo que sigue veremos un caso real de un software de comunicaciones al que le aplicaremos un modelo de Markov. Este modelo se aplica a la arquitectura del sistema a fin de obtener predicciones de performance (tiempo que tarda cada mensaje en atravesar el sistema) y tamaños de buffers que garanticen operatoria en tiempo real sin overflow.

Por el tipo de solución elegida, se debe resolver un caso de comunicación, concurrencia y multithreading en tiempo real.

8.2 Descripción del problema

Se requiere un software de comunicaciones que opere como gateway entre dos brokers internacionales. Por razones de confidencialidad no se darán sus nombres reales, los llamaremos TT y DJ.

Cada uno posee un protocolo de comunicaciones propietario a nivel de enlace, y el sistema debe interactuar por un lado con el host de TT que envía cotizaciones y transacciones financieras desde New York, y por el otro enviarlas a DJ USA.

El mayor problema que afecta al sistema es que debe mantener en tiempo real dos enlaces de comunicación asincrónicos, con distintas velocidades de reconocimiento, control de congestión, tiempos de respuesta, etc. La gran cantidad de subredes y tecnologías intermedias involucradas incrementa la complejidad. (ver fig. 8.1).

La performance también es un factor crítico pues se compite con otros proveedores y quién primero muestra las cotizaciones en las terminales de todo el mundo es normalmente elegido por los operadores bursátiles para la compra - venta de papeles.

8.3 Descripción de los protocolos

8.3.1 TT Feed Specification

Los protocolos son similares a SDLC y HDLC (ver [BLACK93]), aunque un poco más simples. Describiremos ambos en líneas muy generales por dos razones:

- A los efectos de este trabajo basta con saber el mecanismo de envío(send)/reconocimiento(ack), los tiempos medios y la distribuición del flujo más que la estructura interna de los mensajes.

- Puesto que se trata de un caso real, por razones de confidencialidad se omiten más detalles.

Se envían paquetes que contienen básicamente información sobre cotizaciones bursátiles, llamados ítems.

Estos ítems pueden ser de control o de datos, distinguiéndose por el tratamiento requerido según cada uno. Los mensajes de datos contienen una estructura determinada que registra el ID del ítem, precios, atributos para pantalla, etc. No se detalla esa información por no ser relevante en este trabajo.

Línea: RS232 (7E1 19200 bps)

Formato: Header + Message + Trailer

Header: SOH + msgType

Message: STX + MsgBody + ETX

Trailer: CheckSum + EOT

msgType: Tipo de mensaje, de control o de datos

MsgBody: ítem

CheckSum: CRC

No hay un protocolo de reconocimiento ni retransmisiones a nivel de enlace.

8.3.2 DJ Feed Specification

Las pantallas de DJ estan diseñadas como páginas con un reticulado de fila por columna. En cada celda se puede escribir cualquier texto.

Línea: RS232 (300 - 19200 bps)

Formato: STX + PageID + RowCol + Text + ETX + BCC

PageID: Id de la página

RowCol: Codificación de la fila y la columna dentro de la página

Text: Texto a mostrar

BCC: CheckSum

Por cada mensaje enviado se debe esperar a recibir un mensaje de ACK del Host, luego del timeout o por recepción de NAK se retransmite.

8.4 Diseño del programa

Es esencial que el diseño del programa sea coherente con el modelo abstracto.

Básicamente se trata de:

- Dos agentes independientes operando en forma asincrónica que atienden a cada uno de los extremos de la comunicación (TT y DJ).
- Una cola circular de mensajes que sirve como medio de comunicación entre ellos, donde el agente TT es productor y el de DJ es consumidor.
- Una Base de Datos para los ítems de TT
- Un Diccionario para la correspondencia entre los ítems de TT y las celdas de la pantalla de DJ.

En la figura 8.1 se muestra el diagrama de clases del sistema (simplificado) y en la figura 8.2 el diagrama de interacción al recibir un mensaje en el extremo de TT.

TTCommHandler: Handler de comunicación con TT, tiene un agente para administrar el port serial, y la base de datos de ítems donde guarda las actualizaciones.

DJCommHandler: Handler de comunicación con DJ, tiene un agente para administrar el port serial, y el mapping entre los ítems de TT y las celdas de DJ, una cola de salida de mensajes (por cada ítem TT puede haber muchos mensajes en el sistema de DJ), y un convertidor de ítems de TT a DJ.

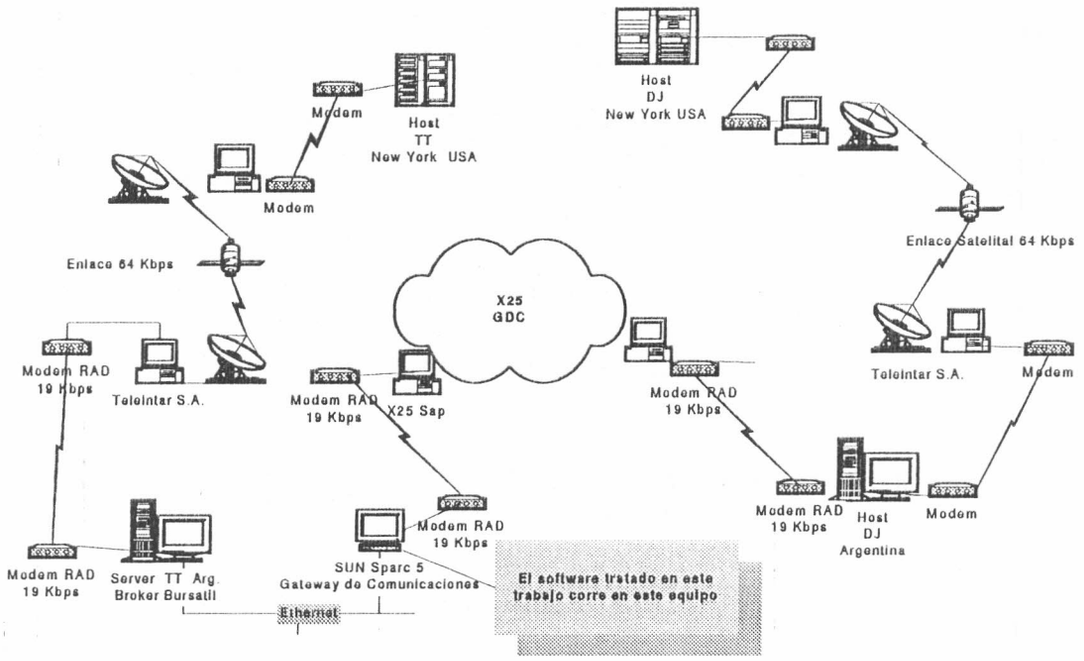
MsgQueue: Cola FIFO que conecta ambos handlers: TT (productor) y DJ (consumidor). Es una cola circular que no tiene inteligencia para controlar su estado. Es responsabilidad de los agentes que la usan evitar el overflow.

DBItems: Base de datos de ítems de TT. Almacena en memoria el estado de cada uno y para algunos accede a disco para guardarlos en una Base de Datos Relacional.

PortHandler: Handler de comunicaciones que gobierna el port de comunicaciones serial.

OutQueue: Cola de salida DJ (formato protocolo DJ).

R4Converter: Convierte una actualización de TT en un conjunto de paquetes en formato DJ para ser enviados a DJ.



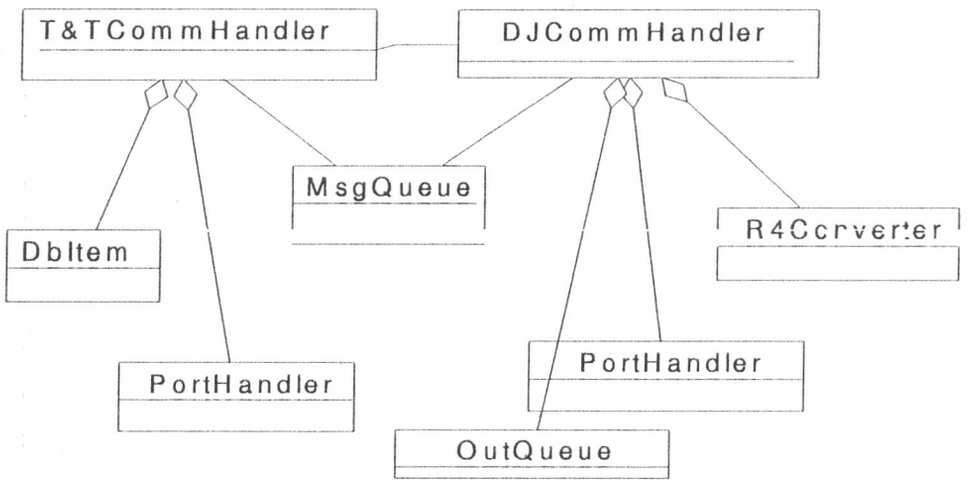
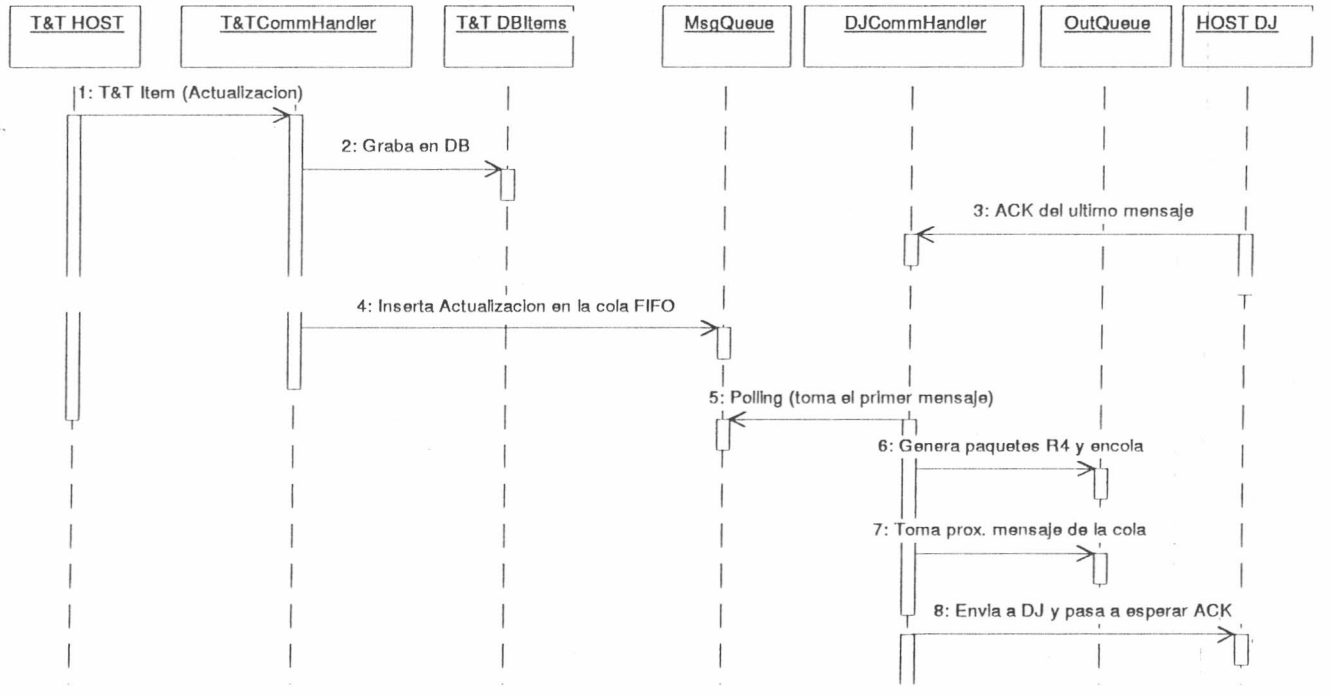


Figura 8.1: Agentes del gateway. Sistema de Comunicaciones

Figura 8.2: Diagrama de Interacción



Capítulo 9

Modelo teórico

9.1 Definiciones de la teoría de colas

En las siguientes secciones utilizaremos la notación de Kendall extendida (ver [HARR93]) para las colas.

Una cola se define con cinco parámetros: A/S/m/N/c, donde:

- **A** describe la naturaleza del proceso de llegada. Por ejemplo, si el proceso de arribo es Poisson A=M (por Markoviano), si es un proceso constante A=D (por determinístico), etcétera.
- **S** describe la distribución del tiempo de servicio. En el caso típico de servicio con distribución exponencial S=M (por Markoviano). Para tiempo de servicio constante S=D (determinístico), y para el caso general S=G.
- **m** indica el número de servidores presentes en el sistema. En nuestro caso m siempre es igual a uno.
- **N** es la capacidad de la cola de espera.
- **c** es la cantidad de clientes posibles en el sistema (en nuestro problema es infinito).

Cuando N y **c** no se especifican, se asume que son infinitos.

9.1.1 Disciplina de la cola

La disciplina de la cola puede ser:

- **FIFO** (First In First Out): Primero en llegar, primero en salir. Es una cola simple común.
- **LIFO** (Last In First Out): Ultimo en llegar, primero en salir. Es el caso de una pila.
- **PS** (Processor sharing) la capacidad de servicio es dividida en forma proporcional entre todos los clientes de la cola. No existe cola propiamente dicha, sino que el tiempo de servicio cambia según la cantidad de clientes existentes (ejemplo: algoritmo round robin).
- **IS** (Infinite Server) se genera un nuevo *clon* del server para cada nuevo cilente.

En este trabajo todas las colas siguen la disciplina FIFO.

9.1.2 Distribuciones utilizadas: Exponencial, Poisson y Coaxian

Para los servidores existen muchas posibles distribuciones. Sin embargo en comunicaciones casi siempre se da el caso de un servidor con distribución exponencial de parámetro λ.

Existe una distribución muy especial llamada Coaxian, que será aplicada en nuestro caso de estudio.

Este modelo fue descrito por Cox (1955) y es conocido como el modelo Coaxian o método de estados.

El modelo consiste en una sucesión de servicios exponenciales de parámetro λ_i, con una probabilidad b_i de no pasar por el servidor i y saltear directamente al próximo estado. Esta distribución es de gran utilidad ya que (ver [HARR93]) cualquier distribución general puede ser aproximada con tanta precisión como se desee por una Coaxian.

9.2 Modelo abstracto

El sistema completo puede representarse como dos colas en tándem: la primera es el agente de TT y la segunda el de DJ, cuya cola es el objeto *MsgQueue* del sistema (ver figura 9.1).

En este tipo de sistemas la llegada de mensajes al sistema normalmente sigue un proceso de Poisson, mientras que el tiempo de servicio de cada agente es Exponencial (ver [MARS96] y [SCHW88]).

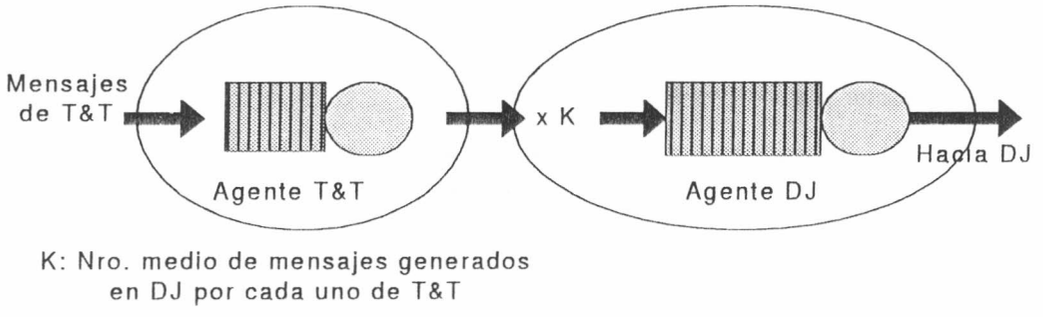


Figura 9.1: Modelo de colas

Se tomaron estas hipótesis, estimándose en cada caso los parámetros:

- *Hipótesis I: Proceso de Ingreso al sistema*
  - $Poisson(\lambda)$

El comportamiento interno de cada agente es el siguiente:

1. *TT*: cada ítem que recibe puede o no ser guardado en una base de datos relacional (dependiendo del ítem) y luego es enviado a la cola de mensajes para DJ.

El modelo elegido para TT es un servidor $Coaxian - 2M/Cox - 2/1/C1 + 1$ con parámetros (s_{11}, s_{12}, Θ_1) donde:

$T_{Enqueue}$ = Tiempo de inserción del msg en la cola de mensajes.

T_{saveDB} = Tiempo que tarda en grabar en la DB cuando hace falta.

Θ_1 = probabilidad de que un msg. deba ser grabado en la DB (obtenida por estimación).

- *Hipótesis II: Servicio TT*
  - $T_{Enqueue}$: $Exponencial(s_{11})$
  - T_{SaveDB} : $Exponencial(s_{12})$

s_{11} s_{12} se obtienen por estimación.

2. *DJ* : Toma los ítems de la cola de mensajes, los transforma en uno o más mensajes de DJ, los encola para enviar, y envía uno por uno.

- *Hipótesis III: Tiempo de servicio DJ*
  - $Exponencial(s_{21})$

s_{21} se obtiene por estimación.

k :número medio de mensajes DJ generados por cada uno de TT.

Otros parámetros :

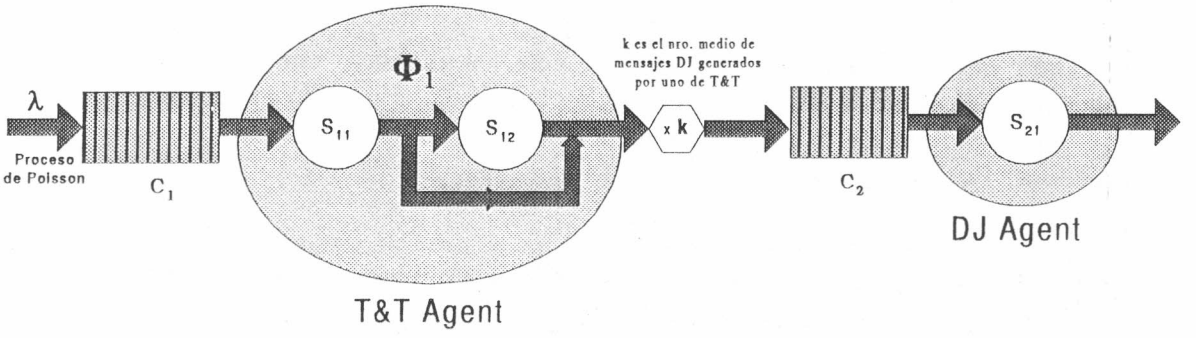
C_1 = tamaño del buffer cola TT
C_2 = tamaño del buffer cola DJ

C_1 y C_2 deberán fijarse de forma tal que se asegure una probabilidad de overflow muy chica (ver 9.4). El modelo completo del sistema a resolver es el de la figura 9.2.

9.3 Análisis de rendimiento

Para resolver el sistema trabajaremos directamente sobre ambas colas deduciendo en forma analítica aproximada los parámetros de interés (es

Figura 9.2: Modelo completo del Gateway



aproximada ya que se realizan aproximaciones a fin permitir el proceso deductivo).

Cabe destacar que en nuestro caso no llegaremos a la generación explícita de la cadena de Markov asociada al sistema de colas puesto que éste puede estudiarse directamente por métodos deductivos sobre cada cola.

La presencia de buffers finitos en un sistema en tándem no permite el tratamiento standard de las colas (como por ejemplo en [SCHW88]) .

La razón es que cuando la segunda cola se bloquea comienza a *rebotar* todo lo que envía la primera, entonces puede mirarse el sistema como una cola con retroalimentación. La tasa media real de ingreso en las colas no será λ sino un valor diferente, y tampoco puede asegurarse que seguirá un proceso de Poisson . Igualmente, haremos la aproximación más usual utilizada entre los investigadores del área (ver [JUN90]) y supondremos que todas las colas son alimentadas por procesos Poisson:

- *Hipótesis IV*
  - Proceso de Ingreso a TT : $Poisson(\lambda_1)$
  - Proceso de Ingreso a DJ : $Poisson(\lambda_2)$

λ_1 y λ_2 son tasas efectivas, desconocidas a priori, y que deberán calcularse.

Vale aclarar que únicamente nos interesa el comportamiento del sistema en estado estacionario, no consideraremos el transitorio.

El método de resolución consiste en aislar cada nodo y analizarlo aproximando las tasas de llegada y partida.

Cuando ocurre bloqueo en la segunda cola, el mensaje bloqueado se encuentra en la primera, pero podemos suponer para el modelo que existe un lugar *ficticio* en cada nodo que ocupa el mensaje bloqueado en el anterior. Por esto el buffer en el nodo 2 tiene capacidad $C_2 + 1$.

Sea n_1 el nodo correspondiente a TT y n_2 el de DJ, llamando fase 1 y fase 2 a los procesos de grabación (s_{11}) y envío (s_{12}) respectivamente del Agente TT, definimos las siguientes probabilidades:

$$\begin{aligned} w_1 &= P(\text{al completarse el servicio en nodo 1 la cola 2 este llena}) \\ w_{1j} &= P(n_1 \text{ en fase j cuando llega un mensaje y encuentra la cola llena}) \\ p_i(n,j) &= P(n_i \text{ tiene n mensajes y el mensaje en servicio en fase j}) \\ p_i(n) &= P(n_i \text{ tiene n mensajes}) = p_i(n,1) + p_i(n,2). \end{aligned}$$

La tasa de arribos al nodo i es λ_i, que es algo mayor que λ, el throughput en estado estacionario. Esto es fácil de ver si consideramos que después de que la cola i está llena los arribos son rechazados.

9.3.1 DJ Agent

En este agente el rendimiento puede analizarse como una cola independiente con buffer finito, tasa de arribo λ_{DJ} y probabilidad de bloqueo P_B:

$$\begin{aligned} \lambda_{DJ} &= k\lambda \\ \text{tasa de msgs rechazados} &= \lambda_{DJ} P_B \end{aligned}$$

El número de mensajes atendidos por unidad de tiempo (throughput) es:

$$\gamma = \lambda_{DJ}(1 - P_B) \tag{9.1}$$

usando el teorema de Little (ver Apéndice A):

$$E(n) = \lambda_{efectiva} E(T)$$
$$E(n) = \lambda_{DJ}(1 - P_B)E(T)$$

$E(T)$: Tiempo medio de un msg. en el sistema.
$E(n)$: Número medio de mensajes en la cola.

$$Tmedio_{DJ} = \frac{E(n)}{\lambda_{DJ}(1 - P_B)} = \frac{\sum_0^\infty np_n}{\lambda_{DJ}(1 - P_B)} \tag{9.2}$$

$$p_n = \frac{(1-\rho)\rho^n}{(1 - \rho^{C_2+1+1})} \tag{9.3}$$

$$\rho = \frac{\lambda_{DJ}}{s_{21}} \tag{9.4}$$

$$\rho = \frac{\lambda k}{s_{21}} \tag{9.5}$$

9.3.2 TT Agent

Sea $S_1^*(s)$ la transformada de Laplace de la función de densidad del servicio original.

$$S_1^* = (1 - \Theta_1)\frac{s_{11}}{s + s_{11}} + \Theta_1 \frac{s_{11}}{s + s_{11}} \frac{s_{12}}{s + s_{12}} \tag{9.6}$$

Si t es el tiempo de servicio efectivo, lo definimos como la suma de otras dos variables aleatorias

$$t = \tau_1 + \tau_2 \tag{9.7}$$

τ_1 = tiempo de servicio original
τ_2 = tiempo de servicio residual cuando hay bloqueo en el nodo 2

La función de densidad del tiempo total de servicio será

$$\begin{aligned} f(t) &= P(\text{no bloqueo}) f(t/\text{no bloqueo}) + P(\text{bloqueo}) f(t/\text{bloqueo}) \\ &= (1 - w_1) f_{Coax-2}(\tau_1) + w_1 f(\tau_1 + \tau_2/\text{bloqueo}) \\ &= (1 - w_1) f_{Coax-2}(\tau_1) + \\ &\quad w_1 P(n_2 \text{ en fase 1}) f(\tau_1 + \tau_2/n_2 \text{ en fase 1}) + \\ &\quad w_1 P(n_2 \text{ en fase 2}) f(\tau_1 + \tau_2/n_2 \text{ en fase 2}). \end{aligned}$$

Sea $B_1^*(s)$ la transformada de Laplace del tiempo efectivo de servicio

$$B_1^*(s) = (1 - w_1)S_1^* + w_1 S_1^* B_2^*(s) \tag{9.8}$$

$$B_1^*(s) = (1 - w_1)S_1^* + w_1 S_1^* \frac{s_{21}}{s + s_{21}} \tag{9.9}$$

Aproximamos esta función por una distribución $Coaxian - 2$ de parámetros $b_1 = (b_{11}, b_{12}, \beta_1)$. Entonces

$$B_1^*(s) \approx (1 - \beta_1)\frac{b_{11}}{s + b_{11}} + \beta_1 \frac{b_{11}}{s + b_{11}} \frac{b_{12}}{s + b_{12}} \tag{9.10}$$

Queremos hallar $(b_{11}, b_{12}, \beta_1)$ como función de w_1 y de los parámetros de n_2. Para esto aplicamos Little a la posición $C_2 + 1$ de n_2

$$p_2(C_2 + 1) = \lambda w_1 E(T_{servM/M/1}) = \frac{\lambda w_1}{s_{21}} \tag{9.11}$$

$$w_1 = \frac{s_{21} p_2(C_2 + 1)}{\lambda} = \frac{s_{21}(1 - \rho)\rho^{C_2+1}}{\lambda(1 - \rho^{C_2+1+1})} \tag{9.12}$$

con w_1 se eligen tres valores de s, reemplazando en 9.9 y asignando a 9.10 se tienen tres ecuaciones para despejar $(b_{11}, b_{12}, \beta_1)$.

Aún queda un parámetro por hallar: la tasa media de ingreso efectiva a n_1. En estado estacionario

$$\lambda_1 = \frac{\lambda}{1 - p_1(C_1 + 1)} \tag{9.13}$$

Esta es una ecuación de punto fijo (ya que es necesario λ_1 para hallar p_1) y la resolvemos con *MathSqueak* aplicando los operadores de punto fijo.

Luego, aplicando Little

$$Tmedio_{TT} = E(T_{TT}) = \frac{1}{b_{11}} + \frac{\beta_1}{b_{12}} \tag{9.14}$$

Finalmente el tiempo total (medio) en el Gateway que toma cada mensaje salido de TT es:

$$T_{Tot} = k\text{Tmedio}_{DJ} + \text{Tmedio}_{TT} \tag{9.15}$$

9.4 Tamaño de los Buffers de Comunicaciones

9.4.1 DJ Agent

Con λ_{DJ} podemos estimar C_2 para una probabilidad de bloqueo dada (P_B). (NOTA: λ_{DJ} es la tasa de entrada multiplicada por k).

Según la *Hipótesis III* cada cola es alimentada por un proceso Poisson, por lo que el problema puede estudiarse separadamente. La cola de DJ nunca se bloquea, por lo que se analiza como un servidor $M/M/1$

En equilibrio:

$$\begin{aligned} \lambda_{DJ} p_n &= s_{21} p_{n+1} \\ p_n &= \rho^n p_0 \\ \rho &= \frac{\lambda_{DJ}}{s_{21}} \\ \sum_{n=1}^{C_2} p_n &= 1. \end{aligned}$$

Resolviendo,

$$p_0 = \frac{1-\rho}{1-\rho^{C_2+1}} \tag{9.16}$$

$$p_{C2} = \frac{(1-\rho)\rho^{C2}}{1-\rho^{C_2+1}}. \tag{9.17}$$

Veamos como ésta es la probabilidad de bloqueo:

$$\begin{aligned} \gamma &= s_{21} P(\text{cola no vacia}) \\ &= s_{21}(1-p_0) \end{aligned}$$

En equilibrio,

$$\begin{aligned} \gamma &= \lambda_{DJ} - \text{tasa de rechazados} \\ &= \lambda_{DJ} - \lambda_{DJ} P_B \\ &= \lambda_{DJ}(1 - P_B.) \end{aligned}$$

Reemplazando por 9.16

$$P_B = p_{C2} = \frac{(1-\rho)\rho^{C_2}}{1-\rho^{C_2+1}} \tag{9.18}$$

Cuando P_B es chico (en nuestro caso 10^{-3})

$$P_B \approx (1-\rho)\rho^{C_2} \tag{9.19}$$

pues,

$$\rho^{C_2+1} \ll 1$$

$$C_2 \approx \log\left(\frac{P_B}{\left(1-\frac{\lambda_{DJ}}{s_{21}}\right)}\right)\frac{1}{\log\left(\frac{\lambda_{DJ}}{s_{21}}\right)} \tag{9.20}$$

9.4.2 TT Agent

El análisis es igual sólo que ahora

$$\rho = \frac{\lambda_1}{E(\text{tiempo serv. Cox-2})} \tag{9.21}$$

$$E(T_{\text{servicio Cox-2}}) = \frac{1}{s_{11}} + \frac{\Theta_1}{s_{s12}} \tag{9.22}$$

$$C_1 \approx \log\left(\frac{P_B}{\left(1-\frac{\lambda_1}{\frac{1}{s_{11}}+\frac{\Theta_1}{s_{12}}}\right)}\right)\frac{1}{\log\left(\frac{\lambda_1}{\frac{1}{s_{11}}+\frac{1}{s_{12}}}\right)} \tag{9.23}$$

Por último, dado que el protocolo de TT soporta control de flujo por software (Xon/Xoff) podemos agregar un control más: cuando se prevea que la cola de mensajes *MsgQueue* se va a llenar enviar un Xoff a TT. Esto es

N_{red} = Nro. msg. de TT en la subred.

T_0 = Tiempo que tarda 1 msg en recorrer la subred

$N_{red} = \lambda T_0$

Desde el envío del Xoff se recibirán aproximadamente

$N_{Tot} = N_{red} + T_0\lambda = 2\lambda T_0$

Haciendo

$C_{1new} = C_2 + \alpha 2\lambda T_0$

con $\alpha > 1$

Cuando la función *MsgQueueCapacity()* de la cola de mensajes (ver Apéndice I) sea igual a

$$MsgQueueCapacity() = \frac{C_1}{C_{1new}}$$

el agente TT deberá enviar un *Xoff*, y cuando *drene* y baje un poco de ese valor, el *Xon*.

Es interesante notar que con este agregado las capacidades estimadas antes darán una tasa real de bloqueo aún menor a la estimada.

Capítulo 10

Implementación

El Gateway corre sobre plataforma SUN Solaris 2.x .

Para la implementación se utilizó ANSI C con standard Posix 1.b para la biblioteca de threads.

Cada agente debe atender la conexión con un Host remoto y a la vez interactuar con la cola de mensajes y el otro agente. El problema que surge es cómo monitorear el handle de cada conexión (que en la abstracción de Unix es un File Descriptor) y encima a la vez realizar otros trabajos. Es decir, como la información proveniente cada Host puede llegar en cualquier momento, cada handle debe estar bloqueado permanentemente esperando algo. En nuestro problema hay dos handlers (uno por cada Host remoto), y realizar un read bloqueante standard no es posible puesto que no se sabe cual será el próximo file descriptor que tenga datos disponibles. ¿ Cómo atender a ambos todo el tiempo ?

En Unix existen 6 alternativas a este problema:

- NonBlocking I/O con polling.
- I/O Asincrónico con el signal SIGPOLL.
- POSIX 1.b I/O asincrónico.
- Usar POLL system call.
- Usar SELECT system call.
- Crear un thread separado para cada conexión.

En [ROB97] pueden hallarse una comparación de las alternativas y ejemplos de cada una.

10.1 Software multithread

La alternativa de varios threads consiste en crear un thread nuevo para atender cada conexión. Se debe proveer un mecanismo para sincronización y control de acceso a datos compartidos. (ver [ROB97] para un tratamiento completo)

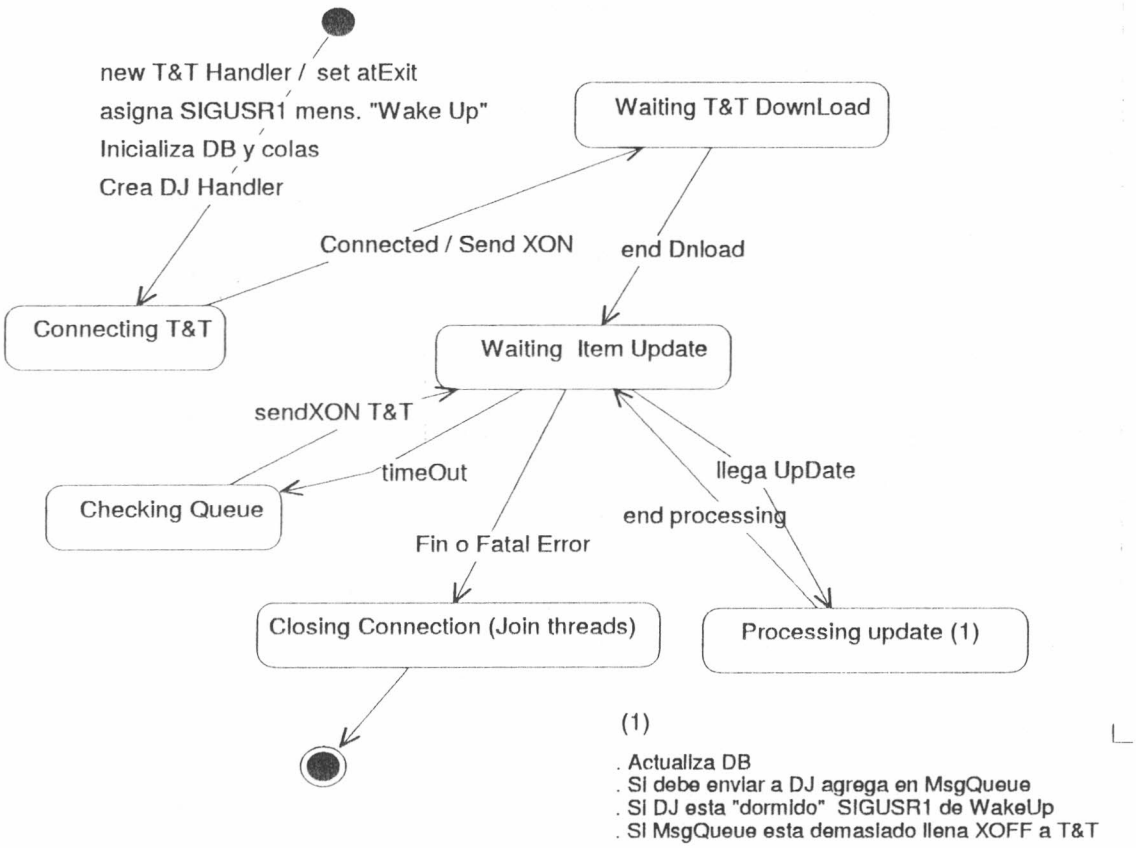
Con esta técnica se puede simplificar el problema pues cada thread se dedica a una sola conexión. Además es posible el procesamiento de datos mientras otra conexión esta siendo leída, es escalable y permite una imlpementación más simple y cercana al modelo abstracto del diseño.

10.2 Implementación de los objetos del sistema

Parte del código C correspondiente a la implementación se incluye en el Apéndice II.

- *TT CommHandler* Atiende la comunicación con el Host de TT mediante el objeto portHandler que contiene. Los paquetes recibidos son pasados a una cola de mensajes que es consumida por el handler de la comunicación con DJ. Realiza el control de flujo (XON / XOFF según el estado de la cola de mensajes MsgQueue).

Figura 10.1: TT CommHandler: transición de estados (simplificado)



- *MSGQueue*
  Implementa la cola circular de mensajes.

  El handler de TT trabaja como productor y el de DJ como consumidor.

  Dado que puede haber accesos simultáneos, se utilizan semáforos (mutex locks POSIX 1.b), para proteger secciones críticas y controlar el acceso concurrente a los datos (ver Apéndice A).

  La cola no se ocupa de controlar su estado. Es un objeto pasivo que responde según el objeto llamador, que debe cuidar de no pisar los mensajes encolados.

  ¿ Por qué se deja a los usuarios de la cola este control tan crítico ?

  Porque es el productor de mensajes quien debe decidir el esquema de control de flujo con el extremo de la red que envía datos (Host de TT). Este agente puede estimar la cantidad de mensajes aún viajando por la subred en cada momento, y enviar el mensaje de control de flujo cuando prevea que la cola será desbordada. En nuestro problema de acuerdo a la tasa de paquetes y el tiempo de viaje de cada uno el control de flujo se hace con de la cola *MsgQueue* llena en la proporción 0.75.

- *DJCommHandler* Se ocupa de la comunicación con DJ. Espera dos eventos asincrónicos: paquetes de DJ, y datos en la cola *MsgQueue.* Estos datos son convertidos a uno o más mensajes para DJ y encolados para ser enviados (*outQueue*) (ver Apéndice C).

  Por cada mensaje proveniente de TT, actualiza la Base de Datos de Items. Si ademas debe enviarlo a DJ lo agrega en la cola *MsgQueue* para que lo tome el handler de DJ.

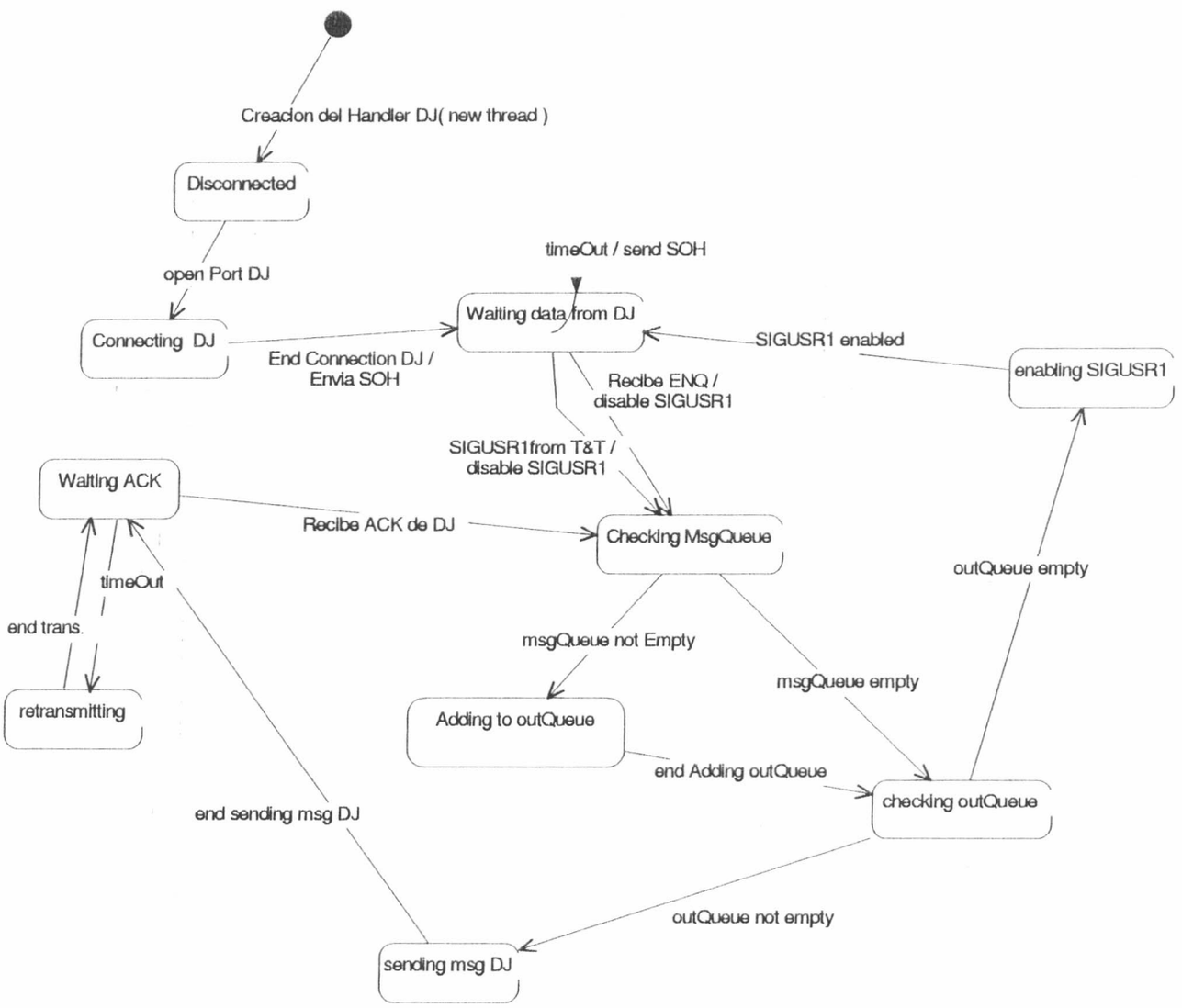
  Cuando la cola está vacía el Handler de DJ solamente está atendiendo la comunicación con su Host; para que se entere que tiene información nueva el agente de TT le envía un mensaje a través de un SIGNAL (SIGUSR1). De esta manera, mientras no haya datos el agente de DJ puede estar *dormido* sin necesidad de monitorear regularmente la cola, y por lo tanto ahorrando recursos de hardware.

- *TT DbItems* Base de datos de los ítems de TT, implementada como una lista doble. No se incluye el código.

- *Port Handler* Handler para interactuar con un port serial. Es una implemenación standard similar a [ROB97] y [STEV90] .

- *OutMsgQueue* Cola FIFO de salida de mensajes hacia el Host de DJ.

Figura 10.2: DJ CommHandler: Diagrama de transición de estados (simplificado)



Capítulo 11

Validación experimental

Para estudiar la validez del modelo teórico se estimaron los parámetros deducidos en el capítulo 9:

- Longitudes de los buffers que garantizan una probabilidad de overflow menor a un umbral fijado
- Tiempo medio de permanencia de cada mensaje que ingresa proveniente de TT.

El primer paso fue estimar mediante intervalos de confianza los parámetros de las distribuciones intervinientes (tasa de ingreso desde TT, probabilidad de que un mensaje sea grabado en la base de datos, tiempo que tarda en recibirse el ACK desde DJ, etc.). Luego, en forma experimental, se obtuvieron los valores reales de las variables en estudio (longitud de las colas, tiempo de permanencia de cada mensaje en el sistema). Finalmente se analizan los resultados experimentales con los obtenidos de las fórmulas al reemplazar las variables por sus estimadores.

No se utilizó el sistema original para el experimento pues actualmente se encuentra en producción. En su lugar se implementó el gateway en un entorno de prueba, desarrollado en MS Visual C++ 5.x y corriendo bajo Windows NT 4.0. Los extremos de TT y DJ se modelaron como fuentes con las distribuciones correspondientes según la estimación realizada en el capítulo 9, y se colocaron en máquinas diferentes conectadas al gateway por una conexión serial (RS232). Con esto el entorno de prueba es análogo al real y permite chequear si la predicción teórica del modelo se refleja en el comportamiento real del software.

11.1 Estimación de los parámetros

Se utilizó la teoría de estimación probabilística para hallar intervalos de confianza. Cabe destacar que la aproximación realizada es válida siempre y cuando las distribuciones supuestas en cada caso correspondan a la realidad.

Dada una variable aleatoria X con media μ y varianza σ^2, para estimar μ a partir de una muestra de n valores se usará la media muestral $\bar{x}$.

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \tag{11.1}$$

Por el Teorema Central del Límite si X tiene distribución Normal o n es suficientemente grande, $\bar{x}$ tendrá distribución Normal; y para $\gamma = 2u - 1$

$$P\left[\bar{x} - \frac{z_u \sigma}{\sqrt{n}} < \mu < \bar{x} + \frac{z_u \sigma}{\sqrt{n}}\right] = \gamma \tag{11.2}$$

Donde γ es un valor cercano a 1 (típicamente 0.97 ó 0.99) y representa la certeza o confiabilidad con que puede afirmarse que el parámetro estimado pertenece realmente al rango, y z_u se obtiene de la tabla Normal(0,1) para $P(x < z_u) = u = (\gamma + 1)/2$.

11.1.1 Distribución Exponencial

En el caso de la distribución Exponencial de parámetro λ como la varianza y la media son interdependientes $\mu = \sigma = \lambda$

$$P\left[\bar{x} - \frac{z_u \lambda}{\sqrt{n}} < \lambda < \bar{x} + \frac{z_u \lambda}{\sqrt{n}}\right] = \gamma \tag{11.3}$$

luego de despejar y reordenar los términos queda

$$P\left[\frac{\bar{x}}{1 + z_u/\sqrt{n}} < \lambda < \frac{\bar{x}}{1 - z_u/\sqrt{n}}\right] = \gamma \tag{11.4}$$

y el intervalo para $\gamma = 2u - 1$ es

$$\frac{\bar{x}}{1 \pm z_u \sqrt{n}} \tag{11.5}$$

11.1.2 Distribución Poisson

Para las variables Poisson de parámetro λ la varianza y la media también son interdependientes $\mu = \sigma^2 = \lambda$

$$P\left[\bar{x} - z_u\sqrt{\frac{\lambda}{n}} < \lambda < \bar{x} + z_u\sqrt{\frac{\lambda}{n}}\right] = \gamma \tag{11.6}$$

Esta relación puede ser escrita de la siguiente forma:

$$P\left[(\lambda - \bar{x})^2 < \frac{z_u^2}{n}\lambda\right] = \gamma \tag{11.7}$$

El intervalo de confianza será aquel que incluya los puntos internos a la parábola:

$$(\lambda - \bar{x})^2 = \frac{z_u^2}{n}\lambda \tag{11.8}$$

Se deben hallar las raíces de esta ecuación cuadrática (λ_1, λ_2) y los puntos comprendidos entre ellas forman el rango del intervalo de confianza.

11.1.3 Probabilidades

Para estimar una probabilidad p se toman n realizaciones del experimento y la estimación es k/n , con k=número de realizaciones que son exitosas.

$$\bar{x} = k/n \tag{11.9}$$

con $\mu_{\bar{x}} = p$ y $\sigma_{\bar{x}}^2 = p(1-p)/n$. Cuando el valor de n es grande vale el Teorema Central y

$$P\left[\bar{x} - z_u\sqrt{\frac{p(1-p)}{n}} < \lambda < \bar{x} + z_u\sqrt{\frac{p(1-p)}{n}}\right] = \gamma \tag{11.10}$$

Como $p(1-p) < 1/4$ en $[0, 1]$ reemplazamos y el intervalo queda

$$\bar{x} \pm \frac{z_u}{2\sqrt{n}} \tag{11.11}$$

11.1.4 Funciones aplicadas a estimadores

Para los casos en los que el parámetro es una función de una o más variables previamente estimadas se utilizó el teorema de Taylor.

Si $y = f(x_1, x_2, ..., x_n)$ y se tienen valores de x_i con cierto error $(x_i \pm \Delta_i)$ entonces el valor estimado de y sera $\bar{y}$ con

$$\bar{y} = f(x_1 + \Delta_1, x_2 + \Delta_2, ..., x_n + \Delta_n) \tag{11.12}$$

Desarrollando por Taylor a primer orden:

$$\bar{y} = f(x_1, x_2, ...x_n) + \nabla f \bullet \vec{\Delta}_{(1,2,..,n)} + o(\Delta^2_{(1,2,..,n)}) \tag{11.13}$$

Despreciando el peso de los términos mayores al primer orden

$$y = \bar{y} \pm \nabla f \bullet \vec{\Delta}_{(1,2,..,n)} \tag{11.14}$$

siendo el error $\nabla f \bullet \vec{\Delta}_{(1,2,..,n)}$.

11.1.5 Resultados predichos teóricamente

En todos los tests se tomó $n = 2000$, $\gamma = 0.99$ y el rango de error obtenido fue menor a un orden de magnitud de la última cifra decimal considerada, por lo que no es incluído en esta sección.

La cantidad media de mensajes de DJ por cada uno de TT (k) es discreto y la estimación se realizó directamente con

$$k = \frac{\sum_{i=1}^{n} i n_i}{n} \tag{11.15}$$

donde n_i es el número de mensajes de TT que generan i de DJ.
Los resultados hallados, con una certeza del 0.99

$$\begin{aligned} k &= 2.00 \pm 0.03 \\ \lambda &= 1.41 \pm 0.006 \\ \lambda_{DJ} &= 2.82 \pm 0.012 \\ s_{11} &= 0.01 \pm 0.002 \\ s_{12} &= 0.42 \pm 0.003 \\ s_2 &= 0.23 \pm 0.003 \\ \Theta &= 0.82 \pm 0.001 \end{aligned}$$

Luego reemplazando en las fórmulas del capítulo 9 se obtiene (para $P_B = 0.01$):

$$\begin{aligned} C_1 &= 11 \text{ msg.} \\ C_2 &= 17 \text{ msg.} \end{aligned}$$

$$\begin{aligned} T_{TT} &= 0.58 \text{ seg.} \\ T_{DJ} &= 0.65 \text{ seg.} \end{aligned}$$

11.2 Resultados experimentales

Experimentalmente se midieron las variables del sistema bajo estudio, es decir: T_{TT}, T_{DJ}, C_1, C_2

En la figura 11.1 se ve la proporción estimada del tiempo total que pasa cada una de las colas en cada estado.

En la figura 11.2 se muestran 3 realizaciones del estudio para el agente TT; se grafica el tiempo de permanencia de cada mensaje dentro del agente.

Los resultados obtenidos son

$$\begin{aligned} E_{TT}(n) &= 6.00 \pm 0.8 \\ E_{DJ}(n) &= 16.32 \pm 1.2 \end{aligned}$$

$$\begin{aligned} C_1 &= 19 \pm 4.3 \text{ msg.} \\ C_2 &= 22 \pm 6.1 \text{ msg.} \end{aligned}$$

$$\begin{aligned} T_{TT} &= 0.64 \pm 0.06 \text{ seg.} \\ T_{DJ} &= 0.70 \pm 0.06 \text{ seg.} \end{aligned}$$

11.3 Validación del modelo

De la figura 11.1 puede verse como la probabilidad de cada estado sigue una curva de decrecimiento exponencial tal como corresponde a este tipo de procesos.

Las medias estimadas se aproximan bien a los valores teóricos, mientras que las cotas para probabilidad menor a 0.99 dieron mayores a lo que predice el modelo.

Otro hecho interesante es que el transitorio en el sistema es casi inexistente. En la figura 11.2 se ve como el tiempo de 'vida' de cada mensaje sigue el mismo patrón desde el comienzo.

En líneas generales, se ve que la implementación refleja el modelo teórico y se comporta según éste predice. Para lograr un mayor grado de exactitud en los resultados se debe analizar con más detalles la estructura interna de cada uno de los procesos involucrados y luego expandir el modelo con otros procesos más *finos* basados también en cadenas de Markov.

Figura 11.1: Estimación de la Longitud de las colas de DJ y TT (normalizada)

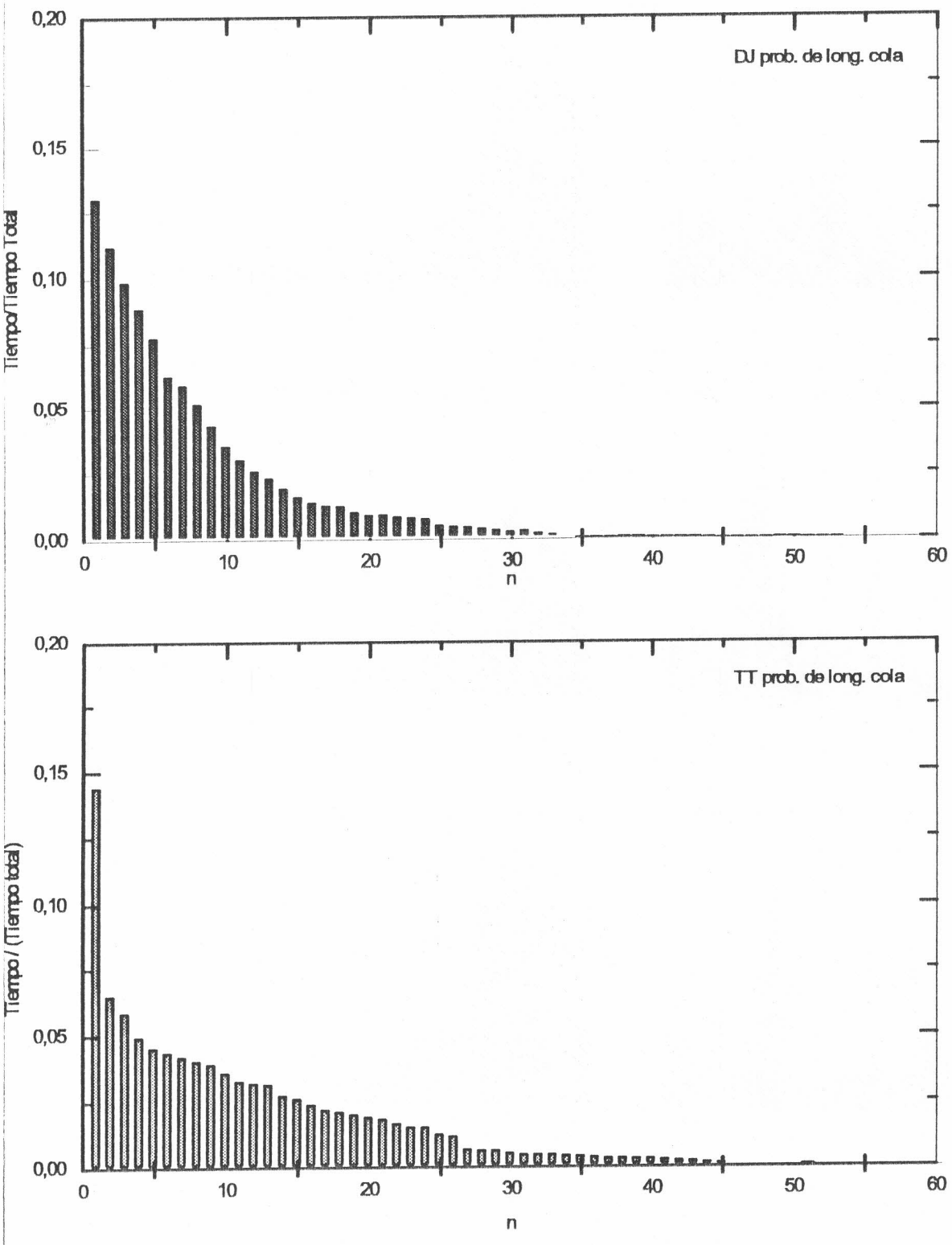
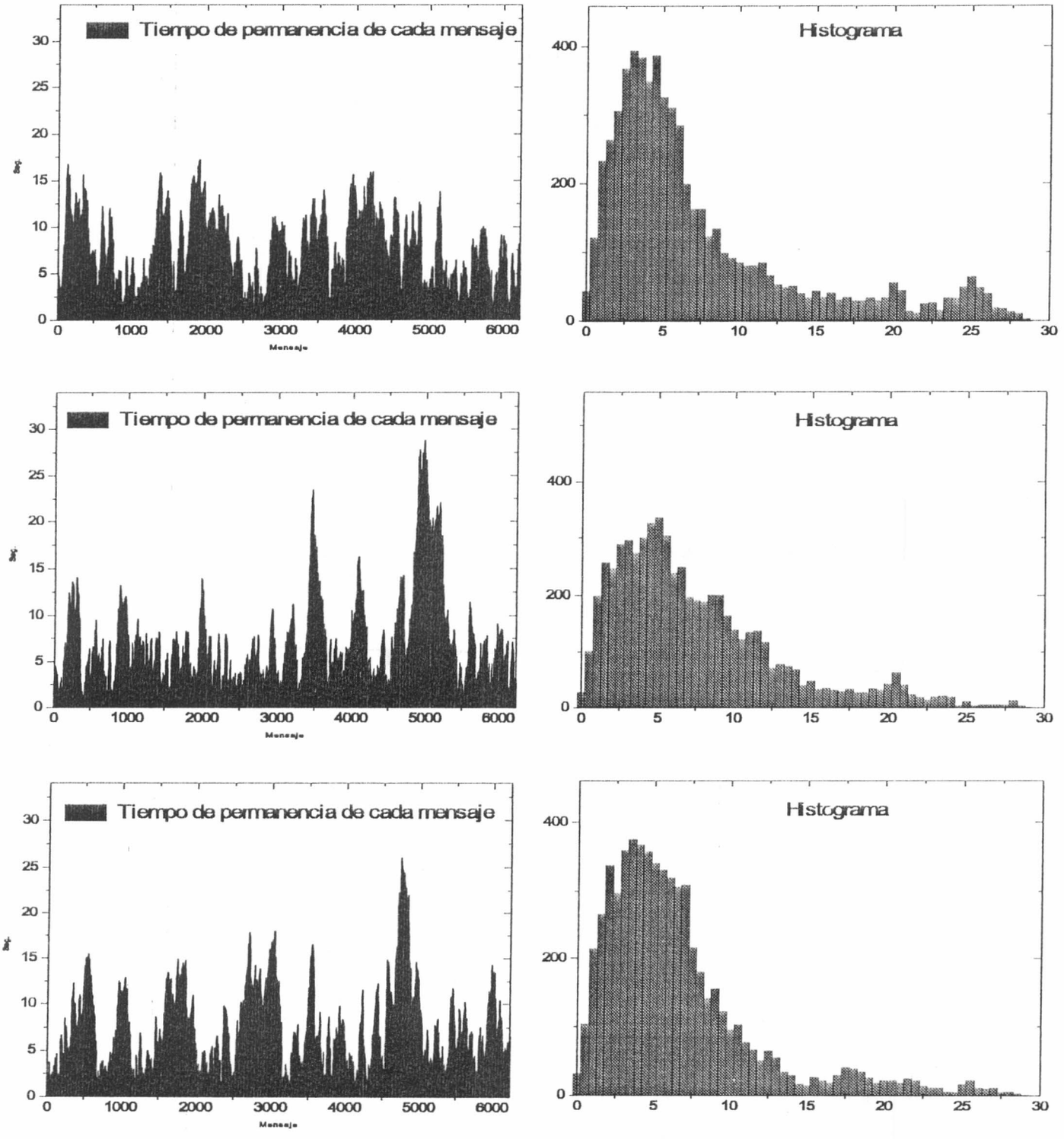


Figura 11.2: Tiempo de permanencia de cada mensaje dentro de TT, con histograma correspondiente



Conclusiones

It is a common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something.
F. D. ROOSEVELT

Squeak es una herramienta asombrosa para respresentar objetos matemáticos y también como entorno de trabajo. Permite una representación flexible y simple, y además el usuario cuenta con todo el poder del lenguaje más puro de objetos, con más de veinte años de desarrollo.

MathSqueak tiene por objetivo ser el punto de partida de un sistema en evolución permanente. Es muy útil para quienes deben resolver sistemas de ecuaciones, cadenas de Markov, estadísticas y simulaciones discretas.

Aún queda mucho por hacer, especialmente en el campo de las cadenas de Markov con millones de estados (actualmente el framework no puede resolver esa clase de problemas). Pero la intención nunca fue dejar un programa cerrado, sino más bien una arquitectura a partir de la cual evolucionar.

Por lo demás, lo mejor de todo el proyecto no es el resultado, sino el hermoso proceso de modelización y representación de la Matemática; el diseño y la creación en Smalltalk con un objetivo ambicioso: que los objetos no implemeten algoritmos, sino que *sean* propiamente los conceptos matemáticos.

Gracias a los threads el diseño y la implementación se simplifican, pues es posible dividir el problema en subproblemas de menor complejidad, es una solución escalable que explota la capacidad de paralelismo del sistema, y permite el aprovechamiento eficiente de los recursos de hardware,

ya que un agente que no tiene trabajo puede *dormirse* hasta ser despertado por cualquier otro que lo alimenta (mediante un signal dirigido). En 30 días de trabajo ininterrumpido se comprobó que el Gateway consumió menos de 15 minutos de tiempo de procesador.

La aplicación de Modelos de Markov al análisis de rendimiento abre un campo muy grande para el estudio de confiabilidad, rendimiento y tolerancia a fallas del software. En particular, se probó en este trabajo que se puede aplicar con éxito la teoría también a la arquitectura misma del software.

Trabajos futuros

It is not your part to finish the task, yet you are not free to desist from it.
Pirke Avoth 2:16
RABBI TARFON, C. 130 C.E.

En MathSqueak existe muchas cosas interesantes para agregar. Algunos ejemplos: estimación de distribuciones, tratamiento de señales discretas, redes bayesianas para IA, métodos de máxima entropía, procesamiento de sonido, respuesta de circuitos lineales a un input estocástico, filtros de Wiener y Kalman, modelos de tráficos en telecomunicaciones, test de hipótesis, redes de Petri estocásticas, etcétera.

Respecto de los sistemas de comunicaciones, estos son algunas líneas de investigación actuales:

Estudio de sistemas en donde los tráficos no pueden ser representados con procesos de Poisson, últimamente se ha encontrado evidencia de que en muchos casos es incluso peor que eso, el tráfico sigue un patrón fractal difícil de modelar (vg. Redes Ethernet, WWW, etc.) ver [CROV97], [FLO95] [CHAOSW]).

En los sistemas de comunicación modernos (real time video, Gigabit Networking, etc.) no sólo el tráfico no es Poisson, sino que además en el estudio por simulaciones muchas veces se requieren recursos inmensos. Nuevas métricas y modelos se deben definir en estos casos para enfrentar los problemas (ver [RTCL]).

Un modelo más detallado del funcionamiento interno del hardware y el sistema operativo donde corre el software. Este mecanismo genera un

sistema de colas bastante más complejo, muchas veces sólo atacable con simulaciones (ver [AGRW87] y [TOL97] para un estudio de este tipo aunque en Base de Datos.)

Sistemas mixtos, donde el grafo que modela el problema tiene algunos nodos con colas normales, y otros que siguen algoritmos determinísticos. En estos casos el estudio teórico es muy complicado (por ejemplo los *Stochastic Network Automata* descritos en [STEW97]).

Las redes de Petri estocásticas han aparecido hace pocos años como una alternativa poderosa a los modelos de colas. Dos razones hacen que hayamos preferido los modelos de Markov en este trabajo: la falta de una teoría fuertemente asentada, y el hecho (ver [MARS96]) que no tienen más poder de expresión que las cadenas de Markov. A su favor está que permiten representar un sistema con un número de estados mucho menor.

Otro campo interesante es el de la ingeniería del software en comunicaciones:

El diseño del software descrito en este trabajo, si bien responde a patrones conocidos de diseño - tomados de la referencia clásica [GAM94]- está diseñado *a pulmón* para el problema. Algunos intentos se han realizado en la búsqueda de un modelo formal automático para la generación y especialmente validación de implementaciones por software de protocolos de comunicaciones. [CHUN94, HUNI95] describen algunas de las interesantes líneas de trabajo, aunque aún no se ha llegado a una técnica madura.

Apéndice A

La fórmula de Little

Consideremos un sistema de una cola como el de la figura A.1 Sea $A(t)$ el número acumulado de arribos en el instante t, y $D(t)$ el número acumulado de salidas que pasan a ser atendidos.

Definimos $L(t) = A(t) - D(t)$. $L(t)$ es el número de mensajes esperando en la cola en el tiempo t. Un estado típico de $A(t)$ puede verse en la figura A.2. Con cada arribo $A(t)$ se incrementa en 1. Mirando la figura podemos definir $n(\tau) = A(\tau) - A(0)$ el número de arribos en el intervalo τ. Entonces la tasa media de arribos en τ es

$$\lambda(\tau) = n(\tau)/\tau \tag{A.1}$$

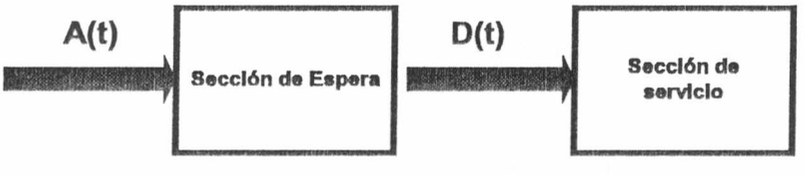
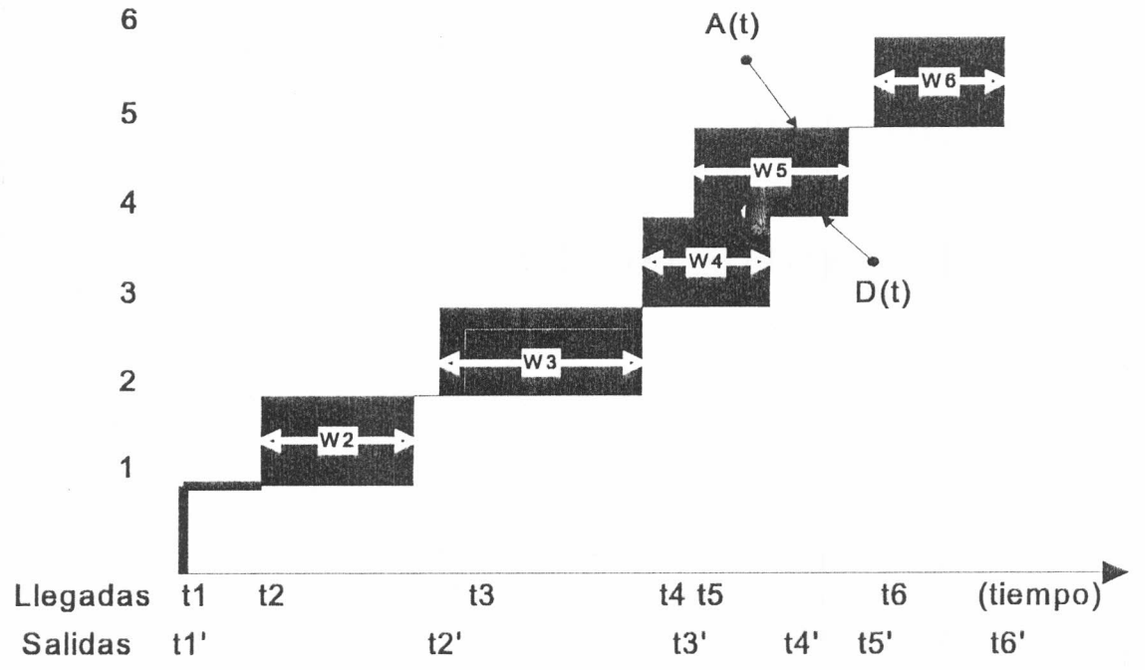


Figura A.1: Little(1)

Figura A.2: Little(2)



Si consideramos ahora las zonas negras de la fig A.2 el área para un intervalo $(0, \tau)$ es

$$\int_0^\tau L(t)dt \tag{A.2}$$

Puesto que $L(t) = A(t) - D(t)$. Sumando los rectángulos de alto 1 y ancho W_j

$$\sum_{j=1}^{n(\tau)} W_j = \int_0^\tau L(t)dt \tag{A.3}$$

Este es el tiempo medio de espera en el intervalo $(0, \tau)$.

Definimos ahora

$$\bar{L}(\tau) = \int_0^\tau L(t)dt/\tau \tag{A.4}$$

Luego

$$n(\tau)\bar{W}(\tau)/\tau = \bar{L}(\tau) = \lambda(\tau)\bar{W}(\tau) \tag{A.5}$$

Esta es la fórmula de Little para disciplina FIFO en $(0\,\tau)$. Si ahora hacemos $\tau, \bar{W}(\tau)\bar{W}, \lambda()\lambda, \bar{L}(\tau)\bar{L}$ y queda

$$\bar{L} = \lambda \bar{W} \qquad \text{A.6)}$$

$\bar{L}$ es el número medio de mensajes en la cola, λ la tasa de arribo, y $\bar{W}$ el tiempo medio de espera en la cola.

Este resultado puede extenderse fácilmente para incluir el tiempo de servicio. También puede probarse que la fórmula no depende de la disciplina de servicio.

Apéndice B

Aplicación: Módem Servers

B.1 Descripción de un Módem Server

Estos sistemas son nodos de Internet que se ocupan de atender las llamadas de los usuarios mediante un pool de módems.

Uno de los problemas que actualmente enfrentan estos programas es la degradación en la performance cuando el sistema escala. Esto es debido a la gran cantidad de procesos (o threads) que operan en paralelo.

La arquitectura del software es decisiva en la performance, distinguiéndose actualmente dos diseños más generales.

B.2 El modelo NFS

Originalmente la arquitectura era la misma que los servidores del sistema de archivo de red de Sun Microsystems (Network File System ó NFS): un proceso (o thread dentro de un programa) que espera nuevas conexiones y cuando recibe una crea un proceso *hijo* que se ocupa de atenderla. Al terminar, el proceso *hijo* muere (ver fig. B.1).

El problema que surge es que a medida que crece el sistema la cantidad de procesos hijos que atienden a los clientes tiende a ser muy grande. El tiempo de procesador asignado a cada uno disminuye y el sistema

consume la mayor parte del tiempo en cambios de contexto (*thrashing*).

B.3 Pool de threads

La otra alternativa empleada actualmente es tener un despachador (*dispatcher*) más un pool de procesos para atención y encolar los pedidos de conexiones cuando están todos ocupados (ver fig B.1).

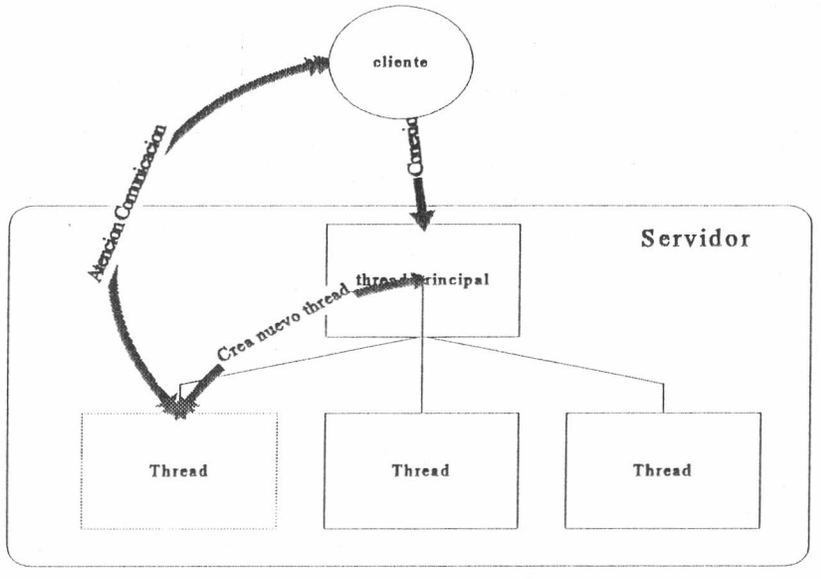
Para pocos procesos simultáneos es más eficiente el primer modelo, mientras que para cantidades muy grandes de conexiones es mejor el segundo.

B.4 ¿ En qué punto conviene cambiar de arquitectura ?

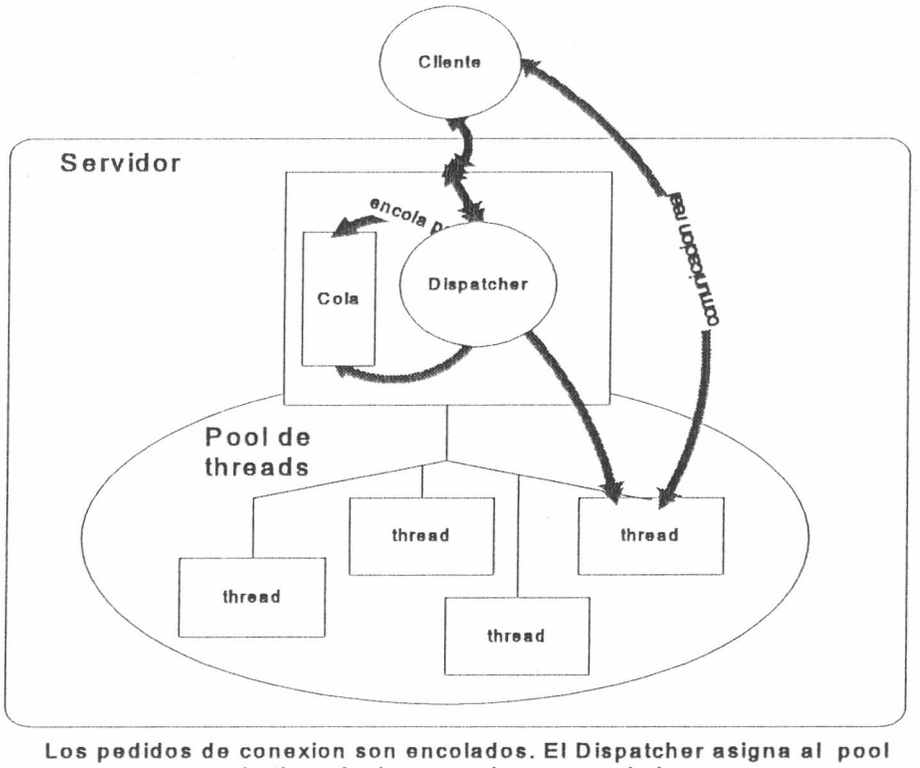
Experimentalmente puede comprobarse que en ciertas condiciones el umbral está en el orden de 20/30 threads. Es decir, el primer diseño es más eficiente si hay pocas llamadas simultáneas y en otro caso el segundo. Cada caso particular tiene un valor de umbral propio.

Un modelo teórico estocástico puede aplicarse para encontrar el punto donde sea más ventajoso pasar de una arquitectura a la otra y en función de ello diseñar e implementar el sistema.

Figura B.1: Arquitecturas para Internet Server



Por cada cliente se crea un nuevo thread que atiende la conexión



Los pedidos de conexion son encolados. El Dispatcher asigna al pool de threads las conexiones emcoladas.

Apéndice C

Una cola circular MT-Safe

La biblioteca standard de C utilizada debe proveer llamadas al sistema reentrantes (read, poll, etc.), es decir que se puedan acceder a la vez por varios threads. Normalemente puede consultarse en el manual del compilador si lo son (Multi Thread Safe).

C.1 Buffers de comunicaciones con colas circulares

Es común que en sistemas de comunicaciones se implementen buffers de entrada/salida mediante colas circulares.

En este Apéndice se incluyen dos casos reales de colas circulares con semáforos para el control de concurrencia.

La primera, desarrollada en ANSI C bajo Unix, es la utilizada en el Gateway tratado en este trabajo.

Utilizando semáforos mutex se restringe el acceso a las zonas críticas a un sólo proceso o thread a la vez. Cada vez que un thread accede a los datos debe lockear el semáforo, si no puede hacerlo queda bloqueado hasta que el mismo sea liberado.

Cuando es posible distinguir accesos de lectura y escritura se pueden utilizar controles menos restrictivos con semáforos de lectura (Read Lock) y otros de escritura (Write Lock) -ver [AGRW87]-. Esta técnica mejora la performance global ya que varios threads pueden leer simultánemente (para escritura el lock es exclusivo).

La implementación de C.1.3 usa esta clase de lock para optimizar la performance. Está desarrollada en ANSI C++ con el standard de tipos paramétricos STL (Standard Template Library) y corre bajo Windows NT 4.x.

C.1.1 Acceso exclusivo (mutex) sobre Unix

Esta es una cola circular de mensajes (de tipo dbItem) que soporta acceso concurrente de múltiples threads gracias a los semáforos mutex (standard POSIX 1b para Unix).

Corresponde al objeto *MsgQueue* del Gateway.

C.1.2 Programa: `msgq.c`

MsgQueue

```
    #define BUFFSIZE      60
#define SLOT_SIZE  256

int  msgQueueInit          ( void     );
int  msgQueueFree          ( void     );
int  msgQueueAdd             ( dbItem * );
int  msgQueueGet           ( dbItem * );
int  numElements           ( void     );
void  msgQueueCapacity         ( double * );

#include "dbItems.h"
#include "msgQueue.h"
#include <pthread.h>

/* semforo para control de acceso */
static pthread_mutex_t  buffer_lock = PTHREAD_MUTEX_INITIALIZER;
/*   buffer y  punteros al principio y fin de la cola */
static dbItem *     buffer [ BUFFSIZE ];
static int          bufOut;
static int          bufIn;
/*=========================================
  numElements: Nro de mensajes encolados
```

```
==========================================*/
int numElements( void )
{   if (bufOut == bufIn || bufOut == -1 )
        return 0;
    return ( BUFFSIZE -- (bufOut - bufIn + BUFFSIZE ) % BUFFSIZE );
}
/*==========================================
  msgQueueInit
==========================================*/
int msgQueueInit( void )
{    int i;

     bufOut =  -1;
     bufIn  =   0;

     for ( i = 0; i< BUFFSIZE; i++ )
          buffer[i] = NULL;
     for ( i = 0; i < BUFFSIZE; i++ )
     {
        buffer[i] = malloc( sizeof(dbItem) );
        if ( ! buffer[i] )
        {    msgQueueFree();
             return ( ERROR );
        }
     }
     return TRUE;
}
/*==========================================
  msgQueueFree
==========================================*/
int msgQueueFree( void )
{    int i;
     for ( i = 0; i < BUFFSIZE; i++ )
        if ( buffer[i] )
            free( buffer[i] );
     return OK;
}
/*==========================================
msgQueueAdd
==========================================*/
int msgQueueAdd ( dbItem * p )
{
  /* Ingresa a la zona critica ------------*/
  pthread_mutex_lock( &buffer_lock );
```

```
    buffer[ bufIn ]→codigo          = p→codigo;
    buffer[ bufIn ]→fechaLastUpdate = p→fechaLastUpdate;

    strncpy( buffer[ bufIn ]→itemLine, p→itemLine,
             MAX_LINE_LENGTH );

    bufIn = (bufIn + 1 ) % BUFFSIZE;

    if ( bufOut == -1 )
     bufOut++;
    /* Sale de la zona critica ------------*/
    pthread_mutex_unlock( &buffer_lock );
    return OK;
}
/*=====================================================
 msgQueueGet
 Copia el contenido del item del buffer en p.
 OJO: p debe estar allocado.
=====================================================*/
int msgQueueGet ( dbItem * p )
{
 /* Ingresa a la zona critica ------------*/
 pthread_mutex_lock( &buffer_lock );

 p→codigo          = buffer[ bufOut ]→codigo;
 p→fechaLastUpdate = buffer[ bufOut ]→fechaLastUpdate;

 strncpy( p→itemLine, buffer[ bufOut ]→itemLine,
          MAX_LINE_LENGTH );

 bufOut = (bufOut + 1 ) % BUFFSIZE;
/* Sale de la zona critica ------------*/
 pthread_mutex_unlock( &buffer_lock );
 return OK;
}
/*=====================================================
 msgQueueCapacity
 devuelve en r la proporcion entre 0 y 1 de cola libre. Sirve al productor
y consumidor para control de flujo con el extremo que manejan..
=====================================================*/
void  msgQueueCapacity( double * r )
{   if (bufOut == bufIn || bufOut == -1 )
      *r = (double) 1;
```

```
    else
        *r = (double) ( (double) ( ( bufOut - bufIn +
                                    BUFFSIZE ) % BUFFSIZE) ) /
                                    (double)  BUFFSIZE;
}
```

C.1.3 Read y Write Locks bajo Windows NT

Esta cola utiliza como estructura de datos para la cola la clase *deque* provista por la biblioteca standard de tipos paramétricos (STL). Para permitir accesos concurrentes por múltiples threads se implementaron locks de lectura y escritura.

Es responsabilidad del agente que accede solicitar un tipo de lock u otro.

El software fue desarrollado en C++ para Windows NT 4.x.

C.1.4 Programa: READWRIT.H

ReadWriteGuard

```
    #ifndef _READWRITEGUARD_H
#define _READWRITEGUARD_H

// ------------------------------------------
// File:          ReadWriteGuard.h
// Description:   clase ReadWriteGuard.
// ------------------------------------------
#include <cassert>
#include "SmartHandle.h"
#include "LockException.h"


class ReadWriteGuard
{
public:
   static const int WAIT_LOCK_TIMEOUT;

   ReadWriteGuard( void );
```

```
    int  getReadCount( void ) const;
    void waitForReadLock( void ) const;
    void waitForWriteLock( void ) const;
    void enterRead( void );
    void leaveRead( void );
    void enterWrite( void );
    void leaveWrite( void );

private:
    int         readCount;
    SmartHandle readMutex;

    SmartHandle writeSemaphore;
};

// ------------------------------------ //
inline int ReadWriteGuard::getReadCount( void ) const
{
    return ( readCount );
}

// ------------------------------------ //
inline void ReadWriteGuard::waitForReadLock( void ) const
{
    if ( WaitForSingleObject( readMutex.get(), WAIT_LOCK_TIMEOUT ) ≠ WAIT_OBJECT_0 )
        throw ( ReadLockException() );
}

// ------------------------------------ //
inline void ReadWriteGuard::waitForWriteLock( void ) const
{
    if ( WaitForSingleObject( writeSemaphore.get(), WAIT_LOCK_TIMEOUT ) ≠
WAIT_OBJECT_0 )
        throw ( WriteLockException() );
}

// ------------------------------------ //
inline void ReadWriteGuard::enterWrite( void )
{
    waitForWriteLock();
}

// ------------------------------------ //
```

```
inline void ReadWriteGuard::leaveWrite( void )
{
    LONG prevCount;

    ReleaseSemaphore( writeSemaphore.get(), 1, &prevCount );

    assert( prevCount == 0 );
}

#endif // _READWRITEGUARD_H
```

C.1.5 Programa: WRITELOC.H

WriteLock

```
    #ifndef _WRITELOCK_H
#define _WRITELOCK_H

// ------------------------------------
// File:          WriteLock.h
// Description:   Declaracion/implementacion de la clase WriteLock,
//                usada para el lock de escritura de una entidad protegida
//                por un ReadWriteGuard.
// ------------------------------------
// Headers
// ------------------------------------ //

#include "ReadWriteGuard.h"


class WriteLock
{
public:
    WriteLock( ReadWriteGuard &_guard, const bool lockEnabled = true );
    ~WriteLock();

    bool isLocked( void );
    void lock( void );
    void unlock( void );

private:
    bool           lockEnabled;
    ReadWriteGuard  &guard;
```

```
};

// ------------------------------------------------------------ //
inline WriteLock::WriteLock( ReadWriteGuard &_guard, const bool lockEnabled ):
    guard( _guard )
{
    if ( lockEnabled )
        lock();

    this->lockEnabled = lockEnabled;
}

// ------------------------------------------------------------ //
inline WriteLock::~WriteLock()
{
    if ( lockEnabled )
        unlock();
}

// ------------------------------------------------------------ //
inline bool WriteLock::isLocked( void )
{
    return ( lockEnabled );
}

// ------------------------------------------------------------ //
inline void WriteLock::lock( void )
{
    guard.enterWrite();
    lockEnabled = true;
}

// ------------------------------------------------------------ //
inline void WriteLock::unlock( void )
{
    guard.leaveWrite();
    lockEnabled = false;
}

#endif // _WRITELOCK_H
```

C.1.6 Programa: READLOCK.H

ReadLock

```
    #ifndef _READLOCK_H
#define _READLOCK_H

// ------------------------------------------
// File:          ReadLock.h
// Description:   clase ReadLock.
//                usada para read locks sobre una entidad protegida por
//                un objeto ReadWriteGuard.
// ------------------------------------------
// Headers
// ------------------------------------------ //

#include "ReadWriteGuard.h"


class ReadLock
{
public:
    ReadLock( ReadWriteGuard &_guard, const bool lockEnabled = true );
    ~ReadLock();

    bool isLocked( void );
    void lock( void );
    void unlock( void );

private:
    bool           lockEnabled;
    ReadWriteGuard  &guard;
};

// ------------------------------------------ //
inline ReadLock::ReadLock( ReadWriteGuard &_guard, const bool lockEnabled ):
    guard( _guard )
{
    if ( lockEnabled )
        lock();

    this->lockEnabled = lockEnabled;
}
```

```
// ------------------------------------------ //
inline ReadLock::~ReadLock()
{
    if ( lockEnabled )
        unlock();
}

// ------------------------------------------ //
inline bool ReadLock::isLocked( void )
{
    return ( lockEnabled );
}

// ------------------------------------------ //
inline void ReadLock::lock( void )
{
    guard.enterRead();
    lockEnabled = true;
}

// ------------------------------------------ //
inline void ReadLock::unlock( void )
{
    guard.leaveRead();
    lockEnabled = false;
}

#endif // _READLOCK_H
```

C.1.7 Programa: LOCKEXCE.H

LockException

```
    #ifndef _LOCKEXCEPTION_H
#define _LOCKEXCEPTION_H

// ------------------------------------------
// File:          LockException.h
// Description:   Declaracion/implementacion de las clases de excepciones
//                disparadas desde el ReadWriteGuard.
// ------------------------------------------

#include <string>
```

```
#include <exception>

using namespace std;


// ------------------------------------ //
// LockException
// ------------------------------------ //

class LockException: public exception
{
public:
   virtual const char *what( void ) const
   {
      return ( "LockException" );
   }
};


// ------------------------------------ //
// ReadLockException
// ------------------------------------ //

class ReadLockException: public LockException
{
public:
   virtual const char *what( void ) const
   {
      return ( "ReadLockException: could not acquire read lock" );
   }
};

// ------------------------------------ //
// WriteLockException - write lock exceptions
// ------------------------------------ //

class WriteLockException: public LockException
{
public:
   virtual const char *what( void ) const
   {
      return ( "WriteLockException: could not acquire write lock" );
   }
};
```

```
#endif // _LOCKEXCEPTION_H
```

C.1.8 Programa: SMARTHAN.H

SmartHandle

```
    #ifndef _SMARTHANDLE_H
#define _SMARTHANDLE_H

// ------------------------------------
// File:          SmartHandle.h
// Description:   Declaracion/implementacion de la clase SmartHandle.
// ------------------------------------ //

#include <windows.h>

// ------------------------------------ //
// SmartHandle - Exception-safe HANDLE container. Llama a closeHandle()
// ------------------------------------ //
class SmartHandle
{
public:
   explicit SmartHandle( HANDLE _handle = NULL );
   SmartHandle( SmartHandle const &smartHandle );
   ~SmartHandle();

   void operator =( SmartHandle const &smartHandle );

   HANDLE get( void ) const;
   HANDLE release( void );
   void   reset( HANDLE _handle = NULL );

private:
   mutable HANDLE handle;
};

// ------------------------------------ //
inline SmartHandle::SmartHandle( HANDLE _handle ):
   handle( _handle )
{
}
```

```
// ------------------------------------------ //
inline SmartHandle::SmartHandle( SmartHandle const &smartHandle ):
    handle( const_cast<SmartHandle &>( smartHandle ).release() )
{
}
// ------------------------------------------ //
inline SmartHandle::~SmartHandle()
{
    if ( handle ≠ NULL )
        CloseHandle( handle );
}
// ------------------------------------------ //
inline void SmartHandle::operator =( SmartHandle const &smartHandle )
{
    reset( const_cast<SmartHandle &>( smartHandle ).release() );
}

// ------------------------------------------ //
inline HANDLE SmartHandle::get( void ) const
{
    return ( handle );
}

// ------------------------------------------ //
inline HANDLE SmartHandle::release()
{
    HANDLE tmp = get();

    handle = NULL;

    return ( tmp );
}

// ------------------------------------------ //
inline void SmartHandle::reset( HANDLE handle )
{
    if ( this→handle ≠ NULL )
        CloseHandle( handle );

    this→handle = handle;
}
#endif // _SMARTHANDLE_H
```

C.1.9 Programa: READWRIT.CPP

ReadWriteGuard

```
    // ------------------------------------
// File: ReadWriteGuard.cpp
//
// Description:  Implementacion de la clase ReadWriteGuard. Provee la
// funcionalidad basica para proteger en forma eficiente cualquier
// entidad que acepta accesos concurrentes de read y write soporta
// simultaneamente multiples reads. Es usada generalemente a traves
// de las clases ReadLock/WriteLock.
// ------------------------------------ //
// Headers
// ------------------------------------ //

#include <windows.h>
#include "ReadWriteGuard.h"
#include "LockException.h"

// ------------------------------------ //
// Static members
// ------------------------------------ //

const ReadWriteGuard::WAIT_LOCK_TIMEOUT = 10000;    // 10 seconds


// ------------------------------------ //
ReadWriteGuard::ReadWriteGuard( void )
{
   readCount = 0;

   readMutex.reset( CreateMutex( NULL, FALSE, NULL ) );
   writeSemaphore.reset( CreateSemaphore( NULL, 1, 1, NULL ) );

   if ( readMutex.get() == NULL )
      throw ( ReadLockException() );

   if ( writeSemaphore.get() == NULL )
      throw ( WriteLockException() );
}
// ------------------------------------ //
void ReadWriteGuard::enterRead( void )
{
```

```
    waitForReadLock();

    if ( ++readCount == 1 )
        waitForWriteLock();

    ReleaseMutex( readMutex.get() );
}

// ------------------------------------------ //
void ReadWriteGuard::leaveRead( void )
{
    waitForReadLock();

    if ( --readCount == 0 )
        ReleaseSemaphore( writeSemaphore.get(), 1, NULL );

    ReleaseMutex( readMutex.get() );
}
```

C.1.10 Programa: RWLOCK.CPP

Programa de test

```
    // ------------------------------------------
// File:          rwlock.cpp
// Description:   Programa que testea las clases ReadWriteGuard, ReadLock
//                y WriteLock (compilar con soporte multithread)
// ------------------------------------------
// Modifications
// ------------------------------------------
// Date       Modified by       Description
// ------------------------------------------
//
// ------------------------------------------

// ------------------------------------------ //
// Headers
// ------------------------------------------ //

#include <windows.h>
#include <cstdlib>
#include <deque>
```

```
#include <iostream>
#include <iomanip>
#include <sstream>
#include <process.h>
#include "ReadWriteGuard.h"
#include "ReadLock.h"
#include "WriteLock.h"

using namespace std;

// ------------------------------------ //
// Constantes
// ------------------------------------ //

const int SHOW_THREAD_COUNT = 5;
const int POP_THREAD_COUNT  = 3;
const int PUSH_THREAD_COUNT = 5;
const int THREAD_COUNT      = SHOW_THREAD_COUNT + POP_THREAD_COUNT +
PUSH_THREAD_COUNT;


// ------------------------------------ //
// variables globales
// ------------------------------------ //

deque<string>       strQueue;
HANDLE              exitEvent          = NULL;
LONG                strNumber          = 0;
ReadWriteGuard      queueGuard;
int                 maxReadLockCount   = 0;
CRITICAL_SECTION    ioSection;

// ------------------------------------ //
// Prototipos
// ------------------------------------ //

unsigned __stdcall  queuePopThread( void * );
unsigned __stdcall  queuePushThread( void * );
unsigned __stdcall  queueShowThread( void * );
BOOL WINAPI         consoleHandler( DWORD ctrlType );


// ------------------------------------ //
int main( void )
```

```
{
   unsigned    threadId[THREAD_COUNT];
   HANDLE      threadHandle[THREAD_COUNT];
   bool        success = true;
   int         i;

   exitEvent = CreateEvent( NULL, TRUE, FALSE, NULL );
   if ( exitEvent ≠ NULL )
   {
      unsigned (__stdcall *threadProc)( void * );

      InitializeCriticalSection( &ioSection );

      SetConsoleCtrlHandler( consoleHandler, TRUE );

      memset( threadHandle, '\0', sizeof ( HANDLE ) * THREAD_COUNT );

      cout ≪ "Creating the " ≪ THREAD_COUNT ≪ " threads that access the deque..."
≪ endl;
      for ( i = 0; i < THREAD_COUNT; i++ )
      {
         if ( i < SHOW_THREAD_COUNT )
            threadProc = queueShowThread;
         else if ( i < SHOW_THREAD_COUNT + POP_THREAD_COUNT )
            threadProc = queuePopThread;
         else
            threadProc = queuePushThread;

         threadHandle[i] = reinterpret_cast<HANDLE>( _beginthreadex( NULL, 0, threadProc,
                           reinterpret_cast<void *>( i ), 0, &threadId[i] ) );

         if ( threadHandle[i] == reinterpret_cast<HANDLE>( -1 ) )
         {
            success = false;
            break;
         }
      }

      if ( success )
      {
         WaitForMultipleObjects( THREAD_COUNT, threadHandle, TRUE, INFINITE );

         cout ≪ "<QUIT> Remaining elements in queue: " ≪ strQueue.size() ≪ "\n"
              "Maximum ReadLock count: " ≪ maxReadLockCount ≪ endl;
```

```
            if ( !strQueue.empty() )
               cout << " [" << strQueue.front().c_str() << "," << strQueue.back().c_str() << "]";

            cout << endl;
         }
         else
            cerr << "ERROR: could not create all the necessary threads\n";

         for ( i = THREAD_COUNT - 1; i >= 0; i-- )
         {
            if ( threadHandle[i] != NULL )
               CloseHandle( threadHandle[i] );
         }

         DeleteCriticalSection( &ioSection );
      }
      else
         cerr << "ERROR: Could not create exit event\n";

      return ( success ? 0 : 1 );
   }


   // ------------------------------------------------------------ //
   unsigned __stdcall queuePopThread( void *threadNumber )
   {
      int i;

      try
      {
         while ( WaitForSingleObject( exitEvent, 0 ) != WAIT_OBJECT_0 )
         {
            WriteLock lock( queueGuard );

            if ( !strQueue.empty() )
            {
               for ( i = 0; i < 200 && !strQueue.empty(); i++ )
               {
                  strQueue.pop_back();
                  strNumber++;
               }

               cout << "<POP " << reinterpret_cast<int>( threadNumber ) <<
```

```
                "> Elements in queue: " << strQueue.size();

            if ( !strQueue.empty() )
               cout << " [" << strQueue.front().c_str() << "," << strQueue.back().c_str() << "]";

            cout << endl;
         }
         else
         {
            cout << "<POP " << reinterpret_cast<int>( threadNumber ) << "> Empty queue" <<
endl;
         }
      }
   }
   catch ( WriteLockException e )
   {
      cerr << "ERROR: WriteLockException caught in queuePopThread() -- " << e.what() <<
"\n"
           "Aborting thread " << reinterpret_cast<int>( threadNumber ) << "\n";
      SetEvent( exitEvent );
   }
   return ( 0 );
}

// ------------------------------------ //
unsigned __stdcall queuePushThread( void *threadNumber )
{
   int     i;
   char    buf[16];

   try
   {
      while ( WaitForSingleObject( exitEvent, 0 ) != WAIT_OBJECT_0 )
      {
         WriteLock lock( queueGuard );

         if ( strQueue.size() < 20000 )
         {
            for ( i = 0; i < 100; i++ )
            {
               strNumber++;
               _itoa( strNumber, buf, 10 );
               strQueue.push_back( string( buf ) );
            }
```

```
            cout << "<PUSH " << reinterpret_cast<int>( threadNumber ) <<
                 "> Elements in queue: " << strQueue.size();

            if ( !strQueue.empty() )
               cout << " [" << strQueue.front().c_str() << "," << strQueue.back().c_str() << "]";

            cout << endl;
         }
      }
   }
   catch ( WriteLockException e )
   {
      cerr << "ERROR: WriteLockException caught in queuePushThread() -- " << e.what() <<
"\n"
           "Aborting thread " << reinterpret_cast<int>( threadNumber ) << "\n";
      SetEvent( exitEvent );
   }
   return ( 0 );
}

// ------------------------------------------------------------ //
unsigned __stdcall queueShowThread( void *threadNumber )
{
   try
   {
      ReadLock lock( queueGuard, false );

      while ( WaitForSingleObject( exitEvent, 0 ) ≠ WAIT_OBJECT_0 )
      {
         // We use an ostringstream
         ostringstream oss;

         lock.lock();

         if ( queueGuard.getReadCount() > maxReadLockCount )
            maxReadLockCount = queueGuard.getReadCount();

         if ( !strQueue.empty() )
         {
            oss << "<SHOW " << reinterpret_cast<int>( threadNumber ) <<
                 "> Elements in queue: " << strQueue.size();

            if ( !strQueue.empty() )
```

```
                oss << " [" << strQueue.front().c_str() << ","
                    << strQueue.back().c_str() << "]\n";

            oss << "--- READ LOCK COUNT: " << queueGuard.getReadCount() << endl;
        }
        else
        {
            oss << "<SHOW " << reinterpret_cast<int>( threadNumber )
                << "> Empty queue" << endl;
        }

        EnterCriticalSection( &ioSection );
        cout << oss.str().c_str();
        cout.flush();
        LeaveCriticalSection( &ioSection );

        lock.unlock();

        Sleep( 0 );
      }
    }
    catch ( ReadLockException e )
    {
      cerr << "ERROR: ReadLockException caught in queueShowThread() -- " << e.what() <<
"\n"
           "Aborting thread " << reinterpret_cast<int>( threadNumber ) << "\n";
      SetEvent( exitEvent );
    }
    return ( 0 );
}


// ------------------------------------ //
// Funcion que es llamada cuando se produce un evento de salida de la
// consola
BOOL WINAPI consoleHandler( DWORD ctrlType )
{
    BOOL    success = TRUE;
    char    *message;

    switch ( ctrlType )
    {
      case CTRL_C_EVENT:
        message = "Ctrl-C detected";
```

```
            break;

        case CTRL_BREAK_EVENT:
            message = "Ctrl-Break detected";
            break;

        case CTRL_CLOSE_EVENT:
            message = "Closing the console";
            break;

        case CTRL_LOGOFF_EVENT:
            message = "The user is logging off";
            break;

        case CTRL_SHUTDOWN_EVENT:
            message = "Shutting down";
            break;

        default:
            success = FALSE;
            break;
    }
    if ( success )
    {
        cout << message << "\n"
             "Please wait while the application is being closed..." << endl;

        SetEvent( exitEvent );
    }
    else
    {
        cerr << "Unknown console event (" << ctrlType << ")\n"
             "The event will be handled by the operating system\n";
    }
    return ( success );
}
```

Bibliografía

[AGRW87] Agrawal, R., and Carey, M. (1987). Concurrency control and Performance Modeling: Alternatives and Implications, *ACM Transactions on Database Systems*, 12, 4, 609-654.

[BACH86] Bach M. (1986). *The Design of the Unix Operating System.* New York: Prentice Hall.

[BIRR89] Birrell A. (1989). *An introduction to programming with threads*, Digital System Research Center.

[BLACK93] Black U. (1993). *Data Link Protocols*, New York: Prentice Hall.

[BURD86] Burden R. (1985). *Análisis Numérico* , México DF: Grupo Editorial Iberoamericano.

[CANI97] Caniglia L. (1997). *Apuntes de la Materia Objetos Matemáticos en Smalltalk* , Departamento de Matemática FCEyN, Universidad de Buenos Aires.

[CHAOSW] The Chaotic Routing Project. Washington University, *http://www.cs.washington.edu /research/ projects/lis /chaos/www /chaos.html*

[COM94] Commer D. E. (1994). *Internetworking with TCP/IP: Volume III: Client-Server Programming and Applications*, New York: Prentice Hall.

[COV91] Cover T., Thomas C. (1991). *Elements of Information Theory*, Reading MA: Addison Wesley.

[CROV97] Crovella M (1997). Self Similarity in World Wide Web Traffic: Evidence and Possible Causes, *IEEE Transactions on Networking*, Vol. 5, Num. 6, 835-846.

[CHUN94] Chung S. (1994). An Object-Based Approach to Protocol Software Implementation, *ACM SIGCOMM* , London UK, 307-316.

[DJ95] D. J. (1995). ** *Gateway Handler Protocol*, Doc No. 40-300.

[FELL57] Feller (1957). *An introduction to Probability Theory and its Applications Vol I*, New York: Wiley.

[FLO95] Floyd S. (1995). Wide area traffic: the failure of the Poisson Model, *IEEE Transactions on Networking.*

[GAM94] Gamma, Helm, Johnson, Vlissides (1994). *Design Patterns. Elements of Reusable Object Oriented Software*, Reading MA: Addison Wesley.

[GOLD89] Goldberg A., and Robson D.(1989). *Smalltalk 80 The Language*, Reading MA: Addison Wesley.

[GOK96] Ggokhale, A., and Schmidt D. (1996). Measuring the Performance of communication Middleware on High-Speed Networks, *ACM SIGCOMM*, Stanford University.

[HARR93] Harrison, P., Patel, N (1993). *Performance Modelling of Communication Networks and Computer Architectures*, Reading MA: Addison Wesley.

[HUNI95] Huni H, et al (1995). A Framework for Network Protocol Software, *OOSPLA*, Austin TX, 358-369.

[ING96] Ingalls D , Kaehler T., Maloney John, Wallace Scott , Kay Alan (1996). *Back to the Future. The Story of Squeak, A Practical Smalltalk Written in Itself,* Walt Disney Imagineering.

[JA95] Jacobson I. (1995). *Object Oriented Software Engineering. A Use Case Driven Approach*, Reading MA: Addison Wesley.

[JUN90] Jun K. P., Perros H.. An approximate analysis of open tandem queing networks with blocking and general services times, *Eur. J. Op. Res.* 46, 123-35.

[KAY93] Kay, A. (1993). The history of Smalltalk, *ACM Sigplan*, New York.

[KING90] King, P (1990). *Computer and Communication System Performance*, New York: Prentice Hall.

[KLEI75I] L. Kleinrock (1975). *Queueing Systems I: Theory* Chichester: John Wiley.

[KLEI75II] L. Kleinrock (1976). *Queueing Systems II: Computer Applications* Chichester: John Wiley.

[KNUT93] Knuth, D. (1993). *The Art of Computer Programming vol II.*, Reading MA: Addison Wesley.

[MARS96] Marsan M. A., et al. (1996). *Modelling with Generalized Stochastic Petri Nets* , New York: John Wiley & Sons.

[MIS93] Mishra B. (1993). *Algorithmic Algebra* , Springer Verlan.

[NIER89] Nierstrasz O. (1989), A Survey of Object Oriented Concepts, en *Object Oriented Concepts, Databases, and aplications*, New York: ACM Press.

[PAP95] Papoulis, A. (1995). *Probability, Random variables and Stochastic Processes*, New York: Addison Wesley.

[PRES92] Press, W (1992). *Numerical recipes in C 2nd ed*, Cambridge UK: Cambridge University Press.

[ROB97] Robins K., Robins N. (1997). *Practical Unix Programming, an introduction to Comunicaction, Concurrency and Multithreading*, New York: Prentice Hall.

[RTCL] Real Time Computing Laboratory , Michigan University, URL: *http://www.eecs. umich.edu/RTCL/*

[RUM91] Rumbaugh J., et al (1991) *Object Oriented Modeling and Design*, New York: Prentice Hall.

[RUB97] Rubino, G. (1997). Técnicas de Análisis de Sistemas de Comunicaciones mediante Modelos de Markov, *Escuela de Ciencias Informáticas 1997*, Buenos Aires: Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires.

[SHAN90] Shanmugan, R (1990). *Randon Signals, detection, estimation and Data Analysis*, Chichester: John Wiley.

[SCHW88] Schwartz, M (1988) *Telecommunications Networks. Protocols, models and data analysis*, Reading MA: Addison Wesley.

[STEW97] Stewart, W. (1997) *Introduction to the Numerical Solution of Markov Chains*, Reading MA: Addison Wesley.

[STEV90] Stevens, R (1990) *Unix Network Programming*, New York: Prentice Hall.

[STEV93] Stevens, R (1993), *Advanced programming in the Unix environment*, Reading MA: Addison Wesley.

[TOL97] Tolomei A. (1997), Estrategias de control de concurrencia de un administrador de Base de Datos. Modelización y Análisis de Performance, *Cátedra de Base de Datos*, Buenos Aires: FCyN UBA.

[TIC98] Tichy, W. (1998). Should computer Scientist experiment more, *IEEE Computer*, Mayo 1998.

[TRI82] K. S. Trivedi. (1982). *Probability and Statistics with Reliability, Queuing and Computer Science Applications.* Chichester: Prentice-Hall.

[TUC89] Tuckwell, H. (1989) *Stochastic Processes in the Neurosciences*, New York: SIAM.

[TT97] TT (1997), *TT Digital Feed Specification*, New York: TT.