Manual de introducción al DDS (Distributed System and Protocols Simulator)

Sandra Moral

Report LSI–96–9–T

Manual de introducción al DSS

(Distributed System and Protocols Simulator)

Sandra Moral Boadas

13 de septiembre de 1996

Índice

Abstract

DSS is a software tool that supports designers in the simulation and analysis of distributed systems and protocols. It is composed of an integrated environment that enables the modelling, description of the topology, creation of the protocol and the simulation of a distributed system. It allows the interaction of the user via an user friendly graphical environment. DSS can be characterized as a sequential discrete event driven simulator. This manual gives a short description of the main components and the basic design methodology followed during its implementation. And finally, it shows the main useful functions that user can use to do its own protocols. These functions do no require sophisticated expertise in new software or language.

Resumen

DSS es una herramienta que ofrece a los diseñadores de algoritmos distribuidos simulaciones y análisis de sus protocolos. Está compuesto por un entorno en el que se integran las funcionalidades de modelar y crear topologías, crear protocolos y simular sistemas distribuidos. Todo esto se ofrece al usuario en un entorno gráfico, que hace que sea fácil y agradable su uso. DSS puede definirse como un simulador secuencial discreto guiado por eventos. Este manual presenta una breve descripción de los componentes principales y comenta la metodología utilizada en el diseño del simulador. Para acabar, aparecen las funciones más útiles para un programador a la hora de realizar sus propios protocolos. Estas funciones no requieren conocimientos de lenguajes sofisticados o programas ya que son muy sencillas.

1 Introducción

Este manual pretende dar al lector una visión global del DSS (Distributed System and Protocols Simulator), profundizando en cómo se realiza la simulación, teniendo en cuenta las estructuras que puede necesitar cualquier programador y las operaciones que puede realizar cualquier usuario.

En el primer apartado se presenta el simulador, ofreciendo una definición acompañada de algunos conceptos (de los que símplemente se pretende repasar). También podemos encontrar una breve descripción de las estructuras utilizadas al implementarlo, así como un poco sobre su origen y sus predecesores. Una vez se conoce el entorno del simulador, se presenta a continuación una descripción más extensa del proceso de simulación, y por último se presentan las topologías y los protocolos, indicándo cómo se pueden crear y qué se puede utilizar para ello.

2 Presentación del sistema

Se puede definir el **DSS como un simulador de sistemas distribuidos que trabaja con una simulación discreta, secuencial y guiada por eventos.** Este apartado explica los conceptos básicos necesarios para comprender esta definición completamente. Además se presenta al DSS dentro de toda una gama de simuladores de algoritmos distribuidos mostrando la ventaja más evidente: la implementación del simulador utilizando un diseño cliente/servidor (que permite aprovechar mejor los recursos disponibles).

No sólo presentamos el DSS en función de los simuladores existentes sino que también se debe presentar dentro del marco de ALCOM-IT como uno de sus proyectos clave en su sección de algoritmos distribuidos.

2.1 Tipos de simulación

Para comprender la definición del DSS es conveniente tener presentes los distintos tipos de simulación que podemos encontrar en cualquier simulador. Una decisión importante al realizar una simulación es decidir en qué momento el sistema debe chequear si tiene algo que hacer, observaciones por tomar, cambios de estado por realizar, etc. Podemos encontrar simuladores diferentes según sea el detonante de este cambio o de este momento de inicio de la actividad, clasificando así las simulaciones como:

- `discretas`, en las que los cambios de estado del sistema son provocados por la llegada de ciertos eventos. Un evento puede ser, por ejemplo, la llegada de un mensaje.
- `contínuas`, en las que las observaciones a recoger o los procesos a realizar se toman siempre cuando tenían previsto hacerse, independientemente al estado del sistema.

2.1.1 Simulación discreta

Dentro de la simulación discreta (guiada por los eventos) podemos encontrarnos dos tipos de simulaciones según se decida tratar los eventos: la secuencial y la distribuida.

En la **simulación secuencial** los eventos se tratan de uno en uno y se guardan en una estructura global (por ejemplo, una lista de eventos) con un reloj global que marca los eventos, y así tratarlos según se vayan produciendo. Este reloj global puede avanzar `guiándose por los eventos` (como los eventos se simulan cronológicamente, el reloj avanza después de cada evento) o bien `guiado por el tiempo` (los eventos se planifican en base a la hora marcada por el reloj que avanza un pulso en cada paso).

En la **simulación distribuida** no existen variables compartidas. Cada máquina trabaja con un proceso y cada proceso trata los eventos que le afectan en el momento en que se producen. El paso de mensajes se realiza entre máquinas.

2.2 Aparición y aplicaciones del DSS

El DSS es una aplicación realizada en el CTI (Computer Technology Institute) de Patras [1], Grecia, dentro del grupo de estudio de aplicaciones de algoritmos distribuidos de ALCOM-IT [2].

Una de las primeras versiones del DSS se instaló en la Universidad de 'La Sapienza' de Roma [3], donde se ha instalado y se ha estado utilizando. También lo utilizan los alumnos de tercero de la Universidad de Patras.

[1] http://www.cti.gr
[2] http://www.mpi-sb.mpg.de/guide/staff/alcom/alcom.html
[3] http://www.dis.uniroma1.it/

Existen versiones comerciales del DSS (en concreto hay dos, la llamada DISYS y la DICOMP), usadas en una empresa griega de telecomunicaciones (INTRACOM) para el diseño y comprobación de sus componentes.

La versión actual del LCLSI es en la que se disponen de protocolos remotos (los que están en el servidor) y protocolos y topologías locales (que son particulares de cada usuario). Además de esta versión con espacios pesonalizados para cada usuario, se prevee que las nuevas versiones incluyan operaciones útiles en la verificación de algoritmos distribuidos. Esta versión funciona sobre la NoWS (Network of Work Stations) con Sun Solaris utilizando las librerias de OSF Motif.

Para entornos en los que no se disponga de terminales gráficos existe una versión con línea de comandos, así como también hay una versión sobre SunOs.

2.3 Requerimientos

Como el código está escrito en C el entorno de ejecución es variable, existen versiones sobre X-Windows y con librerias de OSF Motif, y también se puede ejecutar la versión de línea de comandos sobre cualquier Unix (ya sea Sun Solaris o Linux), ya que simplemente se debe recompilar y no necesita librerias gráficas.

La herramienta para entrorno gráfico ocupa 0'5 Mbytes de código fuente y 1'5 Mbytes el ejecutable. En cuanto a la versión de línea de comandos el espacio que ocupa es 1'2 Mbytes el ejecutable y 0'4 Mbytes el código fuente.

2.4 Estructura de su implementación

La implementación del sistema sigue la siguiente estructura basada en la metodología cliente/servidor.

El sistema se compone de un servidor que atiende las peticiones de tantos clientes como sea necesario. Si las peticiones de los clientes requieren una atención continuada (como sería el caso de pedir una simulación) está pensado que el servidor lance el proceso de simulación y quede libre para servir otras peticiones de otros clientes o del mismo.

El programa cliente se encarga de la presentación del entorno gráfico de la aplicación. Cuando el usuario realiza una petición se encarga de comunicarse con el servidor para que le sirva la petición. Estas peticiones serán sobre topologías y protocolos (creación, modificación, ejecución, borrado, etc). Si el usuario quiere realizar una simulación, el cliente realiza la petición al servidor (pasándole el protocolo y la topología sobre la que se desea simular), y el programa servidor se encarga de compilar el código del simulador junto al del protocolo, lanzando el proceso de simulación y quedando libre para servir otras peticiones (ver Figura 1).

Ventajas de usar una arquitectura cliente/servidor: la ventaja más importante del uso de esta arquitectura es que ofrece un aprovechamiento más eficaz de los recursos disponibles ya que con este sistema podemos utilizar un servidor para varios clientes. Incluso podemos pensar en instalar el servidor en una máquina potente (que sea rápida) y utilizar máquinas más simples para los clientes.

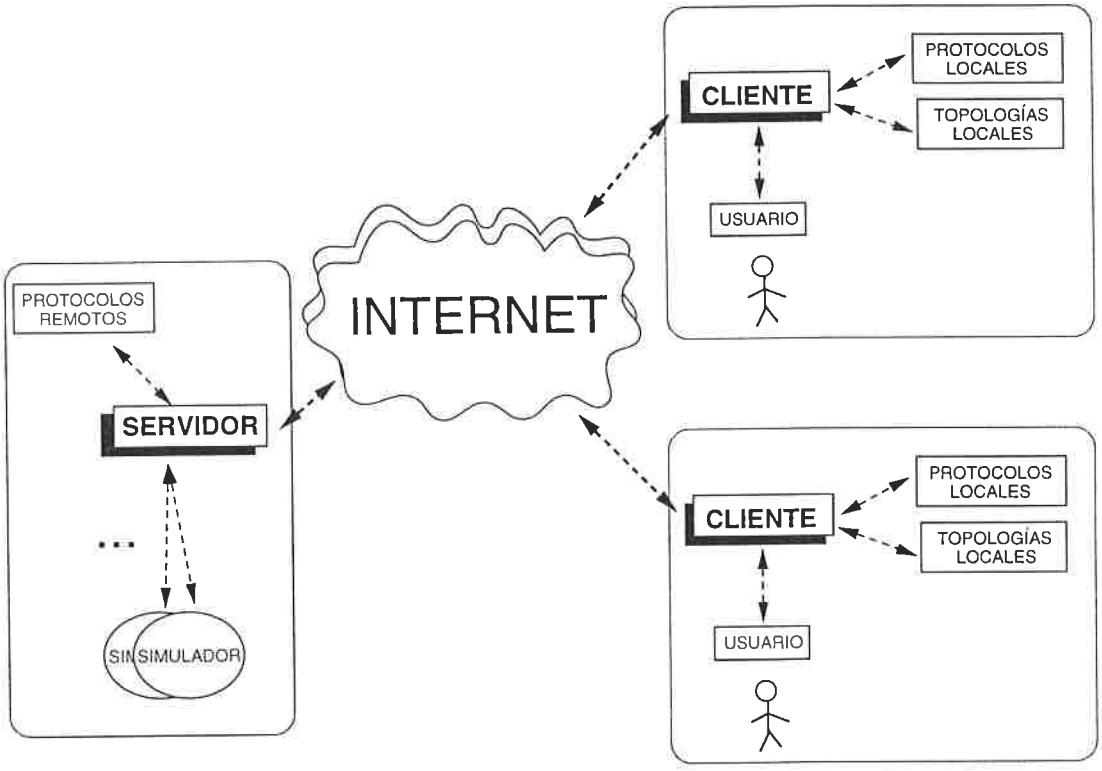

Figura 1: Distribución de los gestores del DSS.

Esta modularidad ofrece muchas ventajas a la hora de reusar o ampliar la aplicación, y como además el lenguaje de implementación de los protocolos es C y no un nuevo lenguaje como ocurre en otros simuladores, hace que el DSS no tenga nada que envidiar a otros sistemas como los que se pueden ver en el siguiente apartado.

2.5 Otros simuladores para algoritmos distribuidos

Existen en el mercado otros simuladores de algoritmos distribuidos, algunos de ellos son:

- PAV (Protocol Analyzer and Verifier) con su lenguaje de especificación, el PSL (Protocol Specification Language) que utiliza una notación CSP, comunicación via mensajes entre máquinas de estados finitos, localiza deadlocks, etc.
- VALIRA (VALidation via Reachability Analysis) define y valida protocolos.
- EDS (Estelle Development System) utiliza Estelle que traduce la descripción del protocolo a C.
- Veda, software de soporte a diseñadores de protocolos.
- DeNet (Discrete Event Network) que tiene un lenguaje de especificación, el DEVS (Discrete Event System Specification) que compila sobre Modula-2.

- SARA (System Architects Apparentice) con su SL (Structure Language).
- NEST System (Network Simulation and Prototyping Testbed) en el que el usuario diseña la topología del sistema, definiendo los parámetros de los nodos y las conexiones (de forma dinámica incluso en tiempo de ejecución) en un entorno gráfico.

3 Modelización del sistema y de los protocolos

Si consideramos el sistema a simular como un conjunto de nodos conectados en una disposición (topología) concreta. Podemos considerar un nodo como un proceso que recibe una secuencia de señales de entrada en un orden determinado. Al recibir una señal de entrada el nodo realiza un conjunto de acciones que dependen de la señal actual y del estado del nodo, pasándo a nuevos estados y/o generándo nuevas salidas. Estos nodos están conectados entre sí con unos canales por los que llegan y salen las señales.

No todo protocolo se puede describir con este tipo de sistemas, por ejemplo un protocolo que controle el número de mesajes transmitiéndose por la red no podría describirse así. Por lo tanto. el sistema se puede modelar así:

$(< \mathbf{S_i} >_{\mathbf{i=1}}^{\mathbf{N}}, < \mathbf{o_i} >_{\mathbf{i=1}}^{\mathbf{N}}. < \mathbf{M_{ij}} >_{\mathbf{ij=1}}^{\mathbf{N}}, \mathbf{succ})$ con N>0, donde

N es el número de procesos.

$< \mathbf{S_i} >_{\mathbf{i=1}}^{\mathbf{N}}$ son N conjuntos finitos disjuntos de S_i, yS_i es el conjunto de estados posibles del proceso i.

$< o_i >_{i=1}^{N}$ es el conjunto de los estados iniciales de cada proceso i.

$< M_{ij} >_{ij=1}^{N}$ son N^2 conjuntos finitos disjuntos de M_{ij}, donde M_{ij} son los mensajes posibles del proceso i al j.

succ función parcial que realiza las transiciones entre procesos. `suc(s,x)` es el estado al que se llega después de enviar/recibir el mensaje `x` en el estado `s`.

Los **protocolos** en el DSS se pueden definir con la siguiente 8-tupla:

$< \mathbf{K}. \mathbf{M}, \mathbf{T}, \mathbf{R}, \mathbf{I}, \mathbf{A}, \mathbf{E}. \mathbf{d} >$ donde

K es el conjunto de estados por los que puede pasar cualquier nodo. (Inicialmente todos los nodos estan en el estado `SLEEPING`.)

M es el conjunto de mensajes que pueden transmitirse entre nodos.

T es el conjunto de relojes (`timers`) que puede utilizar cada nodo.

R es el conjunto de registros (variables) locales que puede usar cada nodo.

I es el conjunto de valores iniciales del sistema. Normalmente son los valores iniciales de las variables locales de cada nodo y los valores de los posibles parámetros de control que utilice el protocolo.

A es el conjunto de acciones asociadas a un estado. En estas acciones se puede enviar mensajes, modificar variables locales y parar o iniciar `timers`.

E es el conjunto de eventos posibles que cada nodo puede recibir o provocar (normalmente serán provocados por la llegada de mensajes o por la finalización de `timers`).

d es la función de transición entre estados. Esta función es de la forma:
$d : (estado, evento) \rightarrow$ acción
$< d >:< K > *(\{INITPROTOCOL\} \bigcup < M > \bigcup < T >) \rightarrow < A >$
con INITPROTOCOL como evento inicial que corresponde al *despertar* de un nodo.

4 Simulación

4.1 Estructuras

Las estructuras a destacar en el momento de la simulación serían:

- **Reloj**, que es un valor real variable que se modifica como en todo simulador guiado por eventos (osea, después de cada evento).
- **Lista de eventos**, esta lista está ordenada por tiempo de ejecución, la podríamos definir como un conjunto de tuplas de la forma (t_i, e_i) donde $t_i \geq$ reloj, osea que el tiempo de la petición es mayor que el tiempo actual. Cada nuevo evento se encola en la lista conservando su t_i.

4.2 Timers

Un nodo dispone de 5 timers (`TIMER_1`, `TIMER_2`,`TIMER_3`, `TIMER_4`, `TIMER_5`). Con los timers se pueden realizar operaciones de forma síncrona, programándolos para que se encargen de contar el tiempo y notifiquen el paso de éste al nodo que les ha programado. Más adelante se pueden encontrar las operaciones que podemos realizar sobre ellos.

4.3 Tipos de eventos

Los eventos que trata el DSS pueden ser producidos por la comunicación entre dos nodos, debido a la notificación del paso de un tiempo determinado por el usuario, o bien porque el sistema informe a un proceso que ya puede iniciar su ejecución. Así pues los eventos que puede tratar el DSS son los siguientes:

- **INITPROTOCOL**, evento inicial de todo nodo. El tiempo de ejecución de este tipo de eventos se toman siguiendo una distribución uniforme (donde los límites se toman del fichero de entrada de inicialización de la topología).
- **RECMES**, evento que indica que se ha recibido un mensaje de un nodo vecino. Cuando este evento ocurre la variable global `CURMESS` contiene toda la información del mensaje.

- **TIMEOUT's**, evento provocado uno de los relojes locales usado con algún fin determinado por el usuario. Este evento informa al nodo que ya ha pasado el tiempo para el que le habían programado.

4.4 Algoritmo del simulador

Se puede ver la simulación como una secuencia de acciones a realizar, este proceso se podría describir con esta secuencia :

1. Tomar el primer evento de la lista.
2. Incluir el evento en la variable global que almacena el evento actual (en el DSS se llama `CUREVENT`).
3. El nodo al que va dirigido el evento realiza las acciones oportunas, cambiando las variables que sean convenientes.
4. Después de realizarse la acción asociada al evento se deben recalcular los tiempos de los otros eventos pentientes. Todo evento puede replanificarse (retrasar su momento de ejecución) excepto los eventos que indiquen que ha pasado un tiempo determinado (como son los `time_out's` de los `timers` programados por los usuarios).

Así pues el simulador del DSS sigue el siguiente algoritmo :

(* Inicialización. *)
clock := 0;
∀ proceso, event_list := (0, INITPROTOCOL);

(* Proceso. *)
`mientras` no se llegue a un criterio de terminación `hacer`
sacar la tupla de menor tiempo de la lista de eventos → `(t,e)`;
simular el efecto de e en el tiempo t;
`fmientras`

La simulación acaba cuando un nodo llama a `simul_end()`.

4.5 Pasos previos a la simulación

Para preparar la simulación debemos seguir los siguientes pasos:

1. Describir formalmente el protocolo que controle el sistema distribuido (en un sistema no simétrico se realizará la asignación de identidades a los nodos).
2. Descripción completa de la topología.
3. Linkaje del protocolo con la topología al simulador.
4. En este momento la simulación está preparada para empezar. Esta simulación se puede visualizar o no en un entorno gráfico, pero, en cualquier caso podemos realizar un fichero con la traza de la simulación.

4.6 Entidades usadas durante la simulación

4.6.1 Gestor de protocolos

Este gestor lee la descripción formal del protocolo generando el código de forma semi-automática.

Las funcionalidades que cubre son:

- Crear protocolos: después de definir completamente el conjunto de estados posibles (y los mensajes), el DSS genera la rutina que controla la transición de estados según los eventos. Las acciones a realizar en las transiciones debe definirlas el usuario y estarán en un fichero C aparte, pero DSS une los dos puntos anteriores en un único fichero.
- Visualizar protocolos.
- Editar protocolos.
- Linkar protocolos con el simulador.
- Borrar protocolos que no sean interesantes para el usuario.

4.6.2 Gestor de topologías

Este gestor nos permite editar/leer las especificaciones necesarias para la creación de nuevas topologías. Para ello se deben seguir los siguientes pasos:

- Usando el ratón, se pueden crear nodos (con sólo un click), enlaces (con un click en cada nodo) o bien se pueden cargar topologías ya hechas y modificarlas. Si se desea ver información sobre la distribución basta con hacer doble click sobre el nodo en concreto.
- Definir características especiales de nodos y enlaces haciendo doble click sobre los nodos.
- Debe comprobarse que la topología sea conexa, y validárse los parámetros de la distribución.

Si el proceso acaba correctamente se generará un bitmap para mostrar la topología en el entorno X-Windows y un fichero ASCII que contiene la conectividad y la distribución de tiempo que debe utilizar el simulador cuando utilice esta topología.

4.6.3 Visualizador de ejecuciones

Realiza la simulación del experimento comunicándose con el módulo de interfície del ûsuario. Para visualizar una ejecución se debe elegir un protocolo y una topología. En ese momento se inicializará el simulador, mostrándose la topología. Una vez aparece la topología en la pantalla se puede ejecutar el protocolo variando parámetros de debug y/o pidiendo las estadísticas que parezcan interesantes.

4.7 Visualización de grandes redes

Se puede definir jerarquías para poder visualizar grandes redes. Si existe una conexión entre los nodos (como mínimo uno) entre dos conjuntos de nodos de un nivel inferior se marca en la visualización con un arco. Se puede ver en detalle un nivel pulsando sobre el nodo que lo representa con el ratón.

5 Estructuras utilizadas

El sistema consiste en un conjunto de nodos conectados entre sí mediante canales por los que se transmiten los mensajes. Para esta transmisión cada nodo puede utilizar sus variables locales, sus variables globales o la información relativa al nodo concreto. Toda esta información está definida en las siguientes estructuras:

- **Estructuras para guardar los mensajes**

  Los mensajes tienen la siguiente estructura C:

```
typedef struct mess {
    int time;
    int drsy-time;
    int kind;
    int value [10];
    long special_field [10];
    int from;
    int hops;
    int maxhops;
    int port;
    struct mess *next;
} MESSAGE;
```

  donde pueden distinguirse *campos de información que ofrece el simulador*:

  - `time`, indica el tiempo de entrega del mensaje,
  - `dray-time`, utilizado para la sincronización,
  - `from`, para saber quién es el originador del mensaje,
  - `hops`, indica el número de nodos por los que ha pasado el mensaje,
  - `port`, indica por qué canal ha llegado,
  - `next`, sirve para mantener la lista de mensajes pendientes del nodo.

  y otros *campos que deben ser modificados por el usuario*, como son:

  - `kind`, que indica el tipo de mensaje que es. Los nombres de los tipos de mensajes son definidos por el diseñador del protocolo.
  - `value`, espacio reservado para transportar un entero. Este campo puede necesitarse o simplemente no tener sentido en algunos protocolos, según lo decida el diseñador.

- `special_field`, son enteros adicionales por si con el campo anterior no es suficiente.
- `maxhops` sirve para indicar el número máximo de nodos por los que puede pasar el mensaje.

- **Estructuras para guardar la información general de un proceso**

  La información general de cada nodo se guarda en una tabla llamada `PCB`. En esta tabla podemos encontrar los siguientes registros :

```
typedef struct {
   int state;
   int id;
   int adjacents;
   PORT *adjust;
   int step-dur [3];
   MESSAGE *mes-buf;
   TIMER clock [5];
   int loc_clock;
   double dr_loc_clock;
   int wake-time;
   int dpq;
   double dr_dpq;
   int step;
   int cycle;
   int idle;
   int init-time;
} PCB ;
```

  donde:

  - `state`, indica el estado actual del nodo,
  - `id`, indica el identificador del nodo,
  - `adjacents`, indica el número de nodos adyacentes al nodo,
  - `adjust`, es una tabla con la información de cada canal con el que se une a los nodos adyacentes. La información que contiene se llama `PORT` y tiene la siguiente estructura:

```
typedef struct {
   int id;
   int delays[3];
   int state;
   int weight;
} PORT;
```

    donde nos indica que estamos conectados al nodo con identificador `id` con un canal que tiene un retardo con parámetros `delays` y que en ese momento está en el estado `state` (para topologías tolerantes a fallos). El campo `weight`

nos indica lo que cuesta transmitir los mensajes, este valor es inicializado por el simulador a un valor constante igual para todos los canales. Si se desea cambiar se debe realizar en la función `init()`,

- `step-dur`, parámetros necesarios para variar la duración de los pasos,
- `mes-buf`, cola de mensajes pendientes por tratar,
- `clock`, indica los timers utilizados,
- `loc_clock`, reloj local del nodo,
- `dr_loc_clock`, error posible en la hora local,
- `wake-time`, es la hora en que empezó a contar el reloj local,
- `dpq`, diferencia de tiempo entre el momento inicial de dos nodos,
- `dr_dpq`, error en la diferencia anterior,
- `step`, indica el paso actual,
- `cycle`, indica el ciclo por el que está el nodo,
- `idle`, indica el momento desde que el nodo esta `idle`[4],
- `init-time`, indica el momento en el que el nodo se despertará.

- **Estructuras para guardar la información local de un proceso.**
  Los datos locales a cada proceso se deberán definir en C, dentro del fichero de acciones del protocolo. Estos datos locales se almacenan en una estructura llamada RESISTERS que tiene esta forma:

```
typedef struct {
    tipo_1 campo_1;
    tipo_2 campo_2;
    ....
} REGISTERS *REGS;
```

  En el fichero de acciones se deben inicializar los valores que queramos en la función `init()`, reservando espacio para la tabla con la rutina siguiente:

```
regalloc()
{
REG= (REGISTERS *)
malloc (processes * sizeof(REGISTERS));
}
```

- **Variables que guardan información global.**

  - `TIME`, reloj global. Para consultarlo se recomienda utilizar la función `get_time()` así evitamos errores en los protocolos.

[4] `idle-time` es el tiempo después de ejecutar una acción durante el cual no se pueden ejecutar otras acciones.

- `PARAM`, tabla que contiene los posibles parámetros del protocolo.
- `me`, indica el identificador del nodo actual.
- `CURMESS`, contiene el mensaje actual.
- `processes`, indica el número de nodos del protocolo.
- `rmin`, `rmax`, `dmin`, `dmax`, indican los valores máximos y mínimos del paso de un nodo y de los retardos de éstos.
- `new-state`, indica el próximo estado del nodo.

6 Topologías

En el entorno de DSS se conoce por topología a la forma como están conectados los `nodos`. Es por estas conexiones llamadas `canales` por dónde se transmiten los mensajes. La transmisión de los mensajes varía según la función de distribución utilizada para calcular el retardo en el canal. La elección de esta función se realiza cuando se crea la topología, y no se puede modificar en tiempo de ejecución. Las funciones que dispone el DSS son:

- **Uniforme**, tendremos que indicar los límites superior e inferior.
- **Geométrica**, hay dos funciones geométricas diferentes (son iguales pero tratan el límite superior de formas diferentes), pero en ambos casos se debe indicar los límites superior e inferior, y el valor principal.
- **Gausaniana**, necesitamos indicar el límite superior, la varianza y el valor inicial (el límite inferior se supone en ambos casos 1).
- **Determinista**, simplemente necesita una constante.

6.1 Principios de sincronización utilizados

Debemos tener en cuenta varios aspectos importantes en la sincronización:

- Cada nodo tiene un reloj local.
- Todos los relojes tienen la misma duración en cada pulsación.
- El retraso en la transmisión de los mensajes es fijo.
- Está permitido procesar más de un evento en un sólo paso.
- Los mensajes recibidos por un nodo son procesados en la siguiente pulsación de reloj local. Esto implica la existencia de memoria local en cada nodo del sistema.
- Los nodos se despiertan de forma aleatoria.

6.2 Proceso de creación de topologías

Una topología para el entorno gráfico del DSS consiste en un grafo donde los nodos son círculos y los canales que unen nodos son la líneas que unen los círculos. A la hora de crear una nueva topología no basta con dibujarla, para realizar una mejor aproximación respecto al comportamiento real de una red el usuario puede elegir para cada canal (o para todos) una función de tiempo de transmisión de los mensajes (con sus correspondientes parámetros). Los pasos a seguir para crear una topología son:

1. Seleccionar dentro del menú principal `TOPOLOGIES` el apartado `CREATE`.

2. Introducir el número máximo de nodos de la nueva topología.

3. Seleccionar la función de distribución de tiempo y sus parámetros. Estos parámetros varian según la distribución del tiempo.

   - Para topologías síncronas :
     - Síncrona (`synchronous`), necesita la duración de un paso bien sea síncrono o ABD[5] pues el retraso será igual en toda la red.
     - Determinística (`deterministic or constant`) el parámetro es una constante.
   - Para topologías asíncronas :
     - Uniforme (`uniform`) indicar el límite superior e inferior.
     - Geométricas (`Geometric1`, `Geometric2`) que necesitan el límite superior, el inferior y el valor de cada paso. La diferencia entre las dos funciones está en el trato del límite superior.
     - Normal (`Normal`, `Gausian`) necesita el valor inicial, el límite superior y la varianza (pues el límite inferior se considera 1).

4. Especificar cuando se producirá el evento de inicialización.

5. Dibujar la topología añadiendo los canales que sean necesarios.

7 Protocolos

7.1 Operaciones disponibles ofrecidas por el sistema

Como se ha visto en anteriormente, podemos definir un protocolo como un autómata, donde en las transiciones se realizan acciones. Para realizar estas acciones, el DSS ofrece un conjunto de operaciones para trabajar con las estructuras ya vistas. Una más o menos breve descripción es la que se puede encontrar en los siguientes apartados.

[5] Asyncronous Bounded Delay, donde los nodos tienen la misma duración del paso pero no van sincronizados.

7.1.1 Operaciones sobre mensajes

Las operaciones siguientes nos permiten el acceso a los mensajes y a las colas donde están almacenados de forma más fácil.

- Manipulación de mensajes en las estructuras.
  - `MESSAGE * create_message()`
    Operación que reserva espacio para un mensaje e inicializa todos sus campos a cero, excepto el campo `from` que lo inicializa con el valor de `me`.
  - `MESSAGE * copy_message(MESSAGE *m)`
    Operación que retorna la copia del pasado, pero sólo de los campos del usuario (`kind`, `hops`, `maxhops`, `value(s)`) y el campo `from` que lo inicializa con el valor de `me`.
  - `insert_queue(MESSAGE **q, MESSAGE *m)`
    Operación que inserta un mensaje en una cola. La inserción se realiza según el campo `time` ya que los mensajes de un mismo proceso no pueden avanzarse en la transmisión. Esta operación es utilizada sobretodo en las operaciones `send_to()`, `broadcast()`, etc.
  - `MESSAGE * extract_queue(MESSAGE **q)`
    Operación que extrae el primer mensaje de la cola. Si la cola esta vacia el puntero que retorna es `NULL`.
  - `insert_queue_last(MESSAGE **q, MESSAGE *m)`
    Inserta el mensaje en la cola situándolo el último. Si para colocarlo tiene que recalcular el tiempo lo hace. También genera el evento `RECMES`.
- Enviar mensajes por la topología.
  Estas operaciones envian mensajes por los canales y siempre que no se indiquen correctamente los canales o bien se intente enviar un mensaje vacío o bien los identificadores de los nodos a los que van dirigidos pueden provocar el paro de la ejecución.
  - `send_to(MESSAGE *m, int p)`
    Operación que envia el mensaje `m` por el canal `p`. La operación lo que hace es colocar el mensaje en la cola de mensajes recibidos `PCB[n].mes_buf` con los parámetros `port` y `hops` actualizados. Esta operación ya se encarga de generar el correspondiente evento `RECMES`. Si el canal por el que se transmite puede fallar se trata también según corresponda.
    En el caso de un anillo se puede utilizar `LEFTPORT` y `RIGHTPORT` para indicar los canales vecinos.
  - `pass_message(int p)`
    Realiza lo mismo que un `send_to()` pero simplemente utilizando como mensaje el `CURMESS`.
  - `send_both(MESSAGE *m)`
    Esta operación sólo se puede utilizar en topología de anillos. y es como si realizase dos `send_to()`. uno a cada vecino.

- `send_to_noadjacent(MESSAGE *m, int n)`
  Operación que envia el mensaje `m` al nodo `n`. Este nodo debe estar conectado con el actual (`me`). El retraso que sufrirá el mensaje es el correspondiente a pasar por todos los nodos que tenga que atravesar para llegar a su destino aumentando convenientemente el campo **hops** hasta que llega al nodo `n`, donde será guardado en la cola de mensajes recibidos y se generará el evento `RECMESS`.
- `broadcast(MESSAGE *m)`
  Operación que envía el mensaje `m` a todos los nodos de la red, sean vecinos o no.
- `port_send(int n)`
  Función que retorna el canal por el que se enviaría un mensaje al nodo `n`.

7.1.2 Operaciones sobre timers

- `start_timer(int t, s)`
  Inicia el timer `t`, planificándose un evento para activarse dentro de `s` pasos. Se marca al **timer** `t` como utilizado (`PCB[me].clock[t].value:=STARTED`). El valor de `t` debe encontrarse entre 1 y 5, o bien ser uno de los valores predefinidos `TIMER1`, `TIMER2`, `TIMER3`, `TIMER4`, `TIMER5`.

- `stop_timer (int t)`
  Operación que anula la operación anterior, eliminando el evento dentro de los eventos planificados, e indicando en el timer `t` que no está utilizado (`PCB[me].clock[t].value:=NOT_STARTED`).

- `stop_timers()`
  Operación que anula todos los timers (del modo que hace la operación anterior).

- `int is_timer()`
  Operación que retorna `TRUE` si hay algún timer activo y `FALSE` en otro caso.

- `int is_a_timer(int t)`
  Función que retorna `TRUE` si el timer `t` está activo.

7.1.3 Operaciones sobre listas de enteros

Estas operaciones son para gestionar listas de enteros. Los elementos que se pueden introducir deben ser menores que el `MAXMEMBERS`.

- `init_list(LIST l)`
  Inicializa la lista `l` a la lista vacia.

- `insert_list(int e, LIST l)`
  Inserta el elemento `e` en la lista `l`. Si el elemento ya estaba en la lista no se modificará.

- `delete_list(int e, LIST l)`
  Borra el elemento `e` de la lista `l`. Si el elemento no estaba en la lista se indicará en el fichero de debug y se mostrará en la pantalla.

- `empty_list(LIST l)`
  Función que retorna `TRUE` si la lista `l` está vacía, y `FALSE` en caso contrario.

- `int in_list(int e, LIST l)`
  Función que retorna `TRUE` si el elemento `e` está en la lista `l`, y `FALSE` en caso contrario.

7.1.4 Operaciones sobre la simulación.

Las operaciones que se presentan a continuación son operaciones específicas de la secuencia de ejecución y también afectan a la información que puede obtener el usuario de cómo se está realizando esta ejecución.

- De control de secuencia.

  - `simul_end()`
    Función que provoca el final de la simulación. Esta función debe ser utilizada alguna vez por algún nodo en toda simulación, ya que de no ser así no acabaría. Una vez finalizada la simulación muestra el mensaje "END OF SIMULATION" por pantalla.

  - `init_event(int n)`
    Función que crea el evento `INITPROTOCOL` y lo planifica con un retraso aleatorio entre los límites `min_init_time`. `max_init_time` que se leen del fichero de entrada de la topología. También existe la función `init_event` que planifica el evento para un instante de tiempo concreto (que se debe indicar en `PCB[i].init_time`)

- De control de posibles errores.

  - `debug(char *options, char *fmt [, arg] ...)`
    Esta función permite obtener información del trascurso de la simulación. También se puede obtener esta información de forma interactiva en el momento de la ejecución. Las opciones posibles para `options` son:

    `-a` para obtener las transiciones de estado de los nodos.

    `-t` para obtener información de los `timers` utilizados.

    `-p` con esta opción el usuario puede introducir comentarios en el fichero de debug.

    `-d` muestra los pasos que se realizan y los retardos.

    `-e` indica los eventos que ocurren.

    `-s` ofrece estadísticas como por ejemplo muestra los mensajes enviados.

    en `fmt` se indica qué formato (en C) se obtendrá la salida, y, `arg` se colocan los parámetros adicionales.

- `simerrror(int stat, int errkind, char *err_mesg)`
  Función que muestra el mensaje de error `err_mesg` siempre que el valor de `errkind` sea `PROTOCOLERROR`. El campo `stat` indica la importancia del error, si su valor es `FATAL` la ejecución se detendrá, pero, si sólo es `WARNING` simplemente se enseña el mensaje.

7.2 Proceso de creación de protocolos

Antes de describir las acciones realizar sobre el terminal, es importante estudiar el protocolo en concreto. Por lo tanto, el proceso de creación de protocolos tiene dos fases:

1. Estudio del protocolo. En esta fase se describen las definiciones necesarias para especificar el protocolo.

2. Transcripción del estudio previo al DSS.

7.3 Descripción de las definiciones necesarias

El nuevo protocolo quedará especificado siempre que se realicen las siguientes acciones:

1. Definir (normalmente con nombres en mayúsculas) :

   - el conjunto de estados posibles por los que podía pasar cualquier nodo,
   - el conjunto de tipos de mensajes que puede enviar un nodo,
   - los `timers` que utilizarán los nodos (si el protocolo es síncrono),
   - las estructuras necesarias para guardar la información propia del protocolo y que no las facilite el DSS.

2. Definir el autómata de transición de estados. Es muy importante saber qué estado es el siguiente después de tratar un evento.

3. Escribir (en C) las acciones a realizar en cada transacción del autómata de estados definido en el punto anterior.

7.4 Breve descripción de las pantallas de definición

Para crear un protocolo en el DSS debemos pulsar la opción `PROTOCOLS` del menú principal. Entonces aparecerá la pantalla de gestión de protocolos, una vez pulsada la opción de creación da paso a la pantalla donde se piden los siguientes datos:

- Breve descripción del protocolo (no más de 80 caracteres).
- Nombre de los parámetros que necesite el protocolo.
- Nombre de los estados (sin poner el `SLEEPING`).
- Nombre de los tipos de mensajes.

- Nombre del fichero C donde esten las acciones a realizar en las transiciones de los estados, la definición de la estructura de las variables locales del nodo (`REGISTERS`), y la función que reserva espacio a esta estructura (`regalloc()`) y la que la inicializa (`init()`). Si este fichero no está editado o se desea editar, el DSS ofrece un pequeño editor.

- El número de `timers` que utilizará cada nodo.

Una vez introducidos todos los parámetros, el DSS va pidiendo los nombres de las acciones a realizar en cada transición provocada por la llegada de un evento en un estado determinado.

Después de esto aparece el código del nuevo protocolo en pantalla, para que el usuario pueda compilarlo hasta corregir todos los errores. Una vez está el código sin errores se puede aceptar dando por terminada la creación del protocolo. Ahora el nuevo protocolo será accesible en la lista de protocolos locales.

7.5 Ejemplos de creación de protocolos

Ejemplo base: *Creación de un protocolo que envia* 1 *mensaje por un anillo unidireccional. El mensaje es generado por el nodo* 0 *y el mismo nodo cuando le llega se encarga de parar la ejecución.*

- Fase de definición de estructuras y tipos :
  Los estados por los que pasa un nodo cualquiera son : `SLEEPING` (estado inicial de todo nodo en todo protocolo en el que estará mientras no haya pasado el mensaje) y `WAITING` (cuando ya haya enviado el mensaje). Sólo se necesita un tipo de mensaje, por ejemplo se puede llamar `PACKET`. No necesitamos ni `timers` ni parámetros adicionales. Tampoco necesitamos guardar datos en el nodo, por lo tanto tampoco tenemos que definir ninguna estructura local.

- Fase de definición de la función de transición será:

  - `SLEEPING x INITPROTOCOL` : Esta transición es posible en todos los nodos, ya que todos reciben un `INITPROTOCOL` pero sólo el nodo cero debe hacer algo (enviar un mensaje a su vecino). *Función C que lo implementa: start()*
  - `SLEEPING x REC_MESS (PACKET)` : Esta transición le llega a todos los nodos pues todo nodo estaba en el estado `SLEEPING` cuando le llega el mensaje, la acción a realizar es pasar el mensaje. *Función C que lo implementa: forward()*
  - `WAITING x INITPROTOCOL` : Este evento es posible para cualquier nodo (menos el nodo 0) ya que pueden haber recibido el mensaje antes de que les despertaran. El nodo 0 jamás puede recibir este evento en este estado porque esto implicaría que tenía dos `INTIPROTOCOL's`. Por lo tanto, la acción a realizar es nula. *Función C que lo implementa: forward ()*
  - `WAITING x REC_MESS (PACKET)` : Esta transición sólo se produce cuando el nodo 0 recibe el mensaje de nuevo. La acción a realizar es finalizar la simulación. *Función C que lo implementa: finished()*

– Escritura en C de las acciones a realizar en cada transición:

```
start()
{
   MESSAGE *mess;
   if (me == 0)
   {
      mess = create_message();
      mes->type = PACKET;
      /* Cada nodo tiene solo un
          canal el 0. */
      send_to(mess,0);
      new_state = WAITING;
    }
}

forward ()
{
   if (me == 0)
      simerror (FATAL, PROTOCOLERROR,"El
      nodo 0 se ha despertado dos veces !");
   else
      pass_message(0);
}

finished ()
{
   simul_end();
}
```

Ampliación 1: *Creación de un protocolo que envia 2 veces un mensaje por un anillo unidireccional. El mensaje es generado por el nodo 0 y el mismo nodo se encarga de parar la ejecución cuando le llega el mensaje por segunda vez.*

- Fase de definición de estructuras y tipos :
  Podemos utilizar los mismos estados, tipos de mensajes y sin necesitar tampoco ningún `timers`. También podemos usar la misma función de transición pero cambiando la función `finish()`, pues ahora no acaba la primera vez que se ejecuta.

- Escritura en C de las acciones que han aparecido y realiza la definición de las variables locales :

```
/* Definicion y reserva de espacio de las
   variables locales. */
typedef struct {int contador;
               } REGISTERS;
REGISTERS *REG;
reg_alloc ()
{
   REG = (REGISTERS *) malloc
            (processes*sizeof(REGISTERS));
}
init ()
{
   int i;
```

```
   for (i=0;i<processes;i++)
      REG[i].contador = 0;
}
finish ()
{
   if (REG[me].contador < 1)
   {
      pass_message(0);
      REG[me].contador ++;
   }
   else
      simul_end();
}
```

Ampliación 2: *Creación de un protocolo que envia k veces un mensaje por un anillo unidireccional. El valor de k se introducirá en tiempo de ejecución y $k > 0$. El mensaje es generado por el nodo 0 y es este mismo cuando le llega el mensaje por k-ésima vez el encargado de parar la ejecución*

- Fase de definición de estructuras y tipos :
  Podemos utilizar los mismos estados, variable local, tipos de mensajes y sin necesitar tampoco ningún `timers`. Tampoco ha cambiado la función de transición pero cambiando la función `finish()` y `init()`, pues ahora acaba la k-ésima vez que se ejecuta.

- Escritura en C de las acciones que ha variado :

```
init ()
{
   int i;
   /* Inicializamos igual el contador. */
   for (i=0;i<processes;i++)
      REG[i].contador = 0;
   /* Lectura del parametro. */
   if (PARAM[0] < 0)
      simerror(FATAL, PROTOCOLERROR,
        "El valor debe ser positivo");
}
finish ()
{
   if (REG[me].contador < PARAM[0])
   {
      pass_message(0);
      REG[me].contador ++;
   }
   else
      simul_end();
}
```

Ampliación 3: *Creación de un protocolo que envía k veces un mensaje por un anillo unidireccional* **síncrono** *El valor de k se introducirá en tiempo de ejecución y $k > 0$. El mensaje es generado por el nodo 0. El protocolo terminará cuando el nodo 0 reciba*

el k-ésimo mensaje o bien cuando hayan pasado m pasos de tiempo desde que empezó la simulación.

- Fase de definición de estructuras y tipos : Podemos utilizar los mismos estados, variable local, tipos de mensajes. Pero ahora sí que es necesario un `timers` para controlar el segundo parámetro del protocolo. La función de transición sí que ha variado, han aparecido dos transiciones más :
  - `SLEEPING x TIMEOUT_1`: Transición es imposible.
  - `WAITING x TIMEOUT_1` : Esta transición puede producirse cuando aún no haya acabado la ejecución y hayan transcurrido más de m pasos de ejecución. *Función C que lo implementa : mi_final()*
- Escritura en C de las acciones que han variado (`start()`, `init()`), y la nueva `mi_final()`:

```
init ()
{
   int i;
   /* Inicializamos igual el contador. */
   for (i=0;i<processes;i++)
      REG[i].contador = 0;
   /* Lectura de los parametros. */
   if (PARAM[0] < 0)
      simerror(FATAL, PROTOCOLERROR,
        "El valor debe ser positivo");
   if (PARAM[1] < 0)
      simerror(FATAL, PROTOCOLERROR,
        "El valor debe ser positivo");
}
start ()
{
   MESSAGE *me;
   if (me == 0)
   {
      start_timer(TIMER1, PARAM[1]);
      mess= create_message();
      mes->type = PACKET;
      send_to(mess,0);
      newstate = WAITING;
}
mi_final ()
{
   simul_end();
}
```