CM

CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS

DEL INSTITUTO POLITECNICO NACIONAL

DEPARTAMENTO DE INGENIERIA ELECTRICA

SECCION DE COMPUTACION

"DISEÑO DE UN LENGUAJE PARA LA COMPUTACION DISTRIBUIDA"

Tesis que presenta el Lic. en Computación Juan José Tevera Mandujano para obtener el grado de MAESTRO EN CIENCIAS en la especialidad de Ingeniería Eléctrica. Trabajo dirigido por el Dr. Jan Janecek Hyan.

Becario del COSNET. México, D.F. junio de 1990.

A MIS PADRES

Sr. Ciro Tevera Jiménez
Sra. Carmen Mandujano Zambrano

Por la confianza depositada en mí.
Por el apoyo y amor que siempre me han brindado.

A MIS HERMANOS

Beatriz, Ciro Abel, Rosa del Carmen y Andrés Rafael

Compañeros fraternales que siempre me han apoyado,
Con cariño a ustedes que han sido mis grandes amigos.

Respetuosamente deseo agradecer al Dr. Jan Janecek Hyan el asesoramiento de este trabajo. Sus comentarios fueron una ayuda inapreciable en el desarrollo del mismo. Agradezco la ayuda brindada por los profesores: Dr. Mario Albarrán Figueroa y al M. en C. César Guzmán Rentería que aparte de los consejos en la elaboración del trabajo hicieron una valiosa revisión de su documentación.

Agradezco al Consejo del Sistema Nacional de Educación Tecnológica (COSNET), por la ayuda brindada para la realización de este trabajo.

Juan José Tevera Mandujano

junio de 1990.

I N D I C E

OBJETIVOS.

Los objetivos de esta tesis es diseñar e implementar un lenguaje cuya finalidad proporcione la ejecución de procedimientos remotos, además de crear el software de comunicación que permita la distribución de la información.

El trabajo se tesis fué dividido en tres partes importantes que son:

1. Diseño del lenguaje distribuido.

   El diseño se enfoca en el lenguaje teórico llamado procesos distribuidos de Hansen.

2. Implementación.

   La implementación se realiza en el sistema Edison

3. Software de comunicación.

   El software de comunicación se realiza en el Edison y MS-DOS.

INTRODUCCION.

En el curso del desarrollo de la computación, importantes avances se han introducido año con año apareciendo soluciones elegantes a un número de problemas. El primer avance fué la computadora misma, seguida por los lenguajes de alto nivel, multiprogramación, sistemas de base de datos, sistemas de información y posteriormente las redes de comunicaciones. Algunos de estos avances fueron previstos en los comienzos de la computación pero se requirió de decadas para iniciar una práctica general. Hoy día, en la parte de procesamiento de datos existe gran importancia hacia la computación distribuída ya que permite solucionar una clase significativa de problemas de una manera natural y efectiva.

Existen aplicaciones importantes en el desarrollo de los lenguajes distribuídos que permiten la distribución de entidades cómo: procedimientos, variables o llamados a procedimientos remotos. Esta última constituye el enfoque que se dá al lenguaje que se presenta en este trabajo de tesis.

El trabajo de tesis está compuesto de cinco capítulos. El capítulo I tiene como objetivo presentar los conceptos y definiciones para el diseño del lenguaje distribuído. Este capítulo está dividido en dos partes: Conceptos de procesos distribuidos de Hansen y Especificación del diseño. La primera parte de este capítulo es la parte medular sobre la concepción de procesos distribuídos, utilizando los conceptos de Hansen. Descripción teórica de los procesos distribuídos. La parte de especificación, presenta las definiciones del diseño basadas en los procesos distribuídos.

El capitulo II, comprende la descripción del lenguaje y el compilador Edison_D. Primeramente se describen las construcciones sintácticas del lenguaje, tratando de no alejarse de la semántica de Edison. Posteriormente se mencionan los

pasos que realiza el compilador para generar el código Edison y se dá mayor énfasis a las primitivas de ambiente distribuido, describiendo la manera como es interpretada por el Kernel.
Este capítulo termina con la descripción de un ejemplo, analizando el código generado y la acción que realiza en el momento de ejecución.

En el capitulo III se crean las primitivas necesarias en Kernel para interpretar el código Edison generado por el compilador Edison_D. Se presenta el mecanismo diseñado para los llamados a procedimientos remotos y se describe el funcionamiento de los procesos definidos en el capítulo I.

En el capítulo IV se presenta el modelo de la red y su implementación en el sistema Edison.. La red lo compone un conjunto de rutinas llamado software de comunicación que permite la transferencia de información mediante el puerto serial.

En el capítulo V se presentan resultados obtenidos de un programa productor-consumidor en forma distribuída. El programa está compuesto de 3 procesos: "producer", "buffer" y "consumer". Cada proceso está definido en las estaciones 1, 2, y 3 respectivamente.

Por último se mencionan las conclusiones obtenidas en el presente trabajo de tesis.

I. DISEÑO.

Este capítulo tiene como objetivo describir el diseño de un lenguaje distribuido.

Primeramente, se realiza una descripción de los conceptos de un lenguaje teórico llamado **procesos distribuidos** de Hansen. Este lenguaje fué introducido por Per Brinch Hansen en el año de 1978 y fué el primer lenguaje en utilizar llamados a procedimientos remotos [Brinch Hansen, 1978].

Para el diseño del lenguaje distribuído se toman los conceptos de este lenguaje y se implementan en el lenguaje Edison, transformando a este último en un lenguaje distribuido tratando de no alejarse de la semántica de Edison [Brinch Hansen, 1981]. De esta manera se pretende dar al programador la facilidad de describir procedimientos de tipo remoto abriendo una posibilidad al lenguaje Edison: La computación distribuída.

I.1 Conceptos de los procesos distribuidos de Hansen.

Los procesos distribuidos de Hansen es un concepto de lenguaje para programación concurrente que se basa en los conceptos de monitor [Hoare, 1974], por medios de procedimientos activos o más bien una colección de procedimientos pasivos. Este lenguaje tiene las siguientes propiedades:

1) Un programa consiste de un número fijo de procesos concurrentes que son inicializados simultáneamente y existirán para siempre. Cada proceso puede accesar únicamente sus propias variables, no existiendo variables comunes entre procesos.
2) Un proceso puede llamar a procedimientos comunes definidos en otros procesos. Estos procedimientos son ejecutados cuando los procesos esperan por alguna condición verdadera. Esta es la única forma de comunicación entre procesos.
3) Los procesos son sincronizados por medio de proposiciones no determinísticas llamadas regiones guardadas.

Las propiedades del lenguaje especifican que un programa estará constituído de un número fijo de procesos concurrentes donde cada proceso puede accesar únicamente sus variables propias.
Estos procesos pueden ser usados como módulos de programas en un sistema multiprocesador con almacenamiento común o distribuido. Así, cuando un proceso espera por alguna condición verdadera entonces el procesador también espera hasta que un procedimiento externo haga la condición verdadera.
Un proceso define sus variables propias, algunos procedimientos comunes y una parte inicial. La forma general de un proceso es la siguiente:

process nombre
variables propias
procedimientos comunes
parte inicial

Un proceso puede llamar a procedimientos comunes definidos en él mismo o en otros procesos. De esta manera un procedimiento que llama de un proceso a otro es denominado una **solicitud externa** y es la única manera en que los procesos se pueden comunicar.

Un proceso realiza dos clases de operaciones: la primera es la parte inicial y la segunda el servicio a la solicitud externa hecha por otros procesos. Estas operaciones son realizadas una a la vez por interacción. Un proceso inicia ejecutando la primera parte, continúa hasta que termina alguna proposición o espera que alguna condición sea verdadera. Entonces la segunda operación inicia como resultado de una solicitud externa. Esta manera de operar (parte inicial y solicitud externa), continúa para siempre.
Si la parte inicial termina, el proceso continúa existiendo aceptando solicitudes externas.
La interacción es controlada por el programa y no por señales de reloj a nivel de máquina. Un proceso pasa de una operación a otra únicamente cuando una operación termina o espera por una condición en la región guardada.
Un proceso continúa la ejecución de operaciones excepto cuando todas sus operaciones se encuentren en regiones guardadas o cuando éste realice una solicitud a otros procesos. En el primer caso, el proceso está ocioso hasta que otro proceso le llame. En el segundo caso, el proceso está ocioso hasta que el otro proceso haya completado la operación solicitada por este.
Un proceso garantiza únicamente que se realizarán las operaciones mientras existan operaciones que puedan ser procesadas, pero únicamente el programador puede asegurar que todas las operaciones se realicen en un tiempo finito.

Un procedimiento define sus parámetros de entrada y salida, algunas variables locales y una parte procedual que es ejecutada cuando éste es llamado.

```
proc nombre (parámetros de entrada, #parámetros de salida)
  variables locales
  parte procedual
```

Un proceso P puede llamar a un procedimiento R definido en otro proceso Q de la manera siguiente:

```
call Q.R (expresiones, variables)
```

Cuando la operación R es realizada, los valores de la expresión de llamado son asignados a los parámetros de entrada del procedimiento. Cuando la operación es finalizada, los valores de los parámetros de salida son asignados a las variables del llamado.
La figura 1.1 muestra un esquema de un programa en procesos distribuídos de Hansen, compuesto de tres procesos X, Y, Z.
La comunucación entre procesos es realizada mediante los llamados **call Y.P1, call Z.P1, call X.P2.**

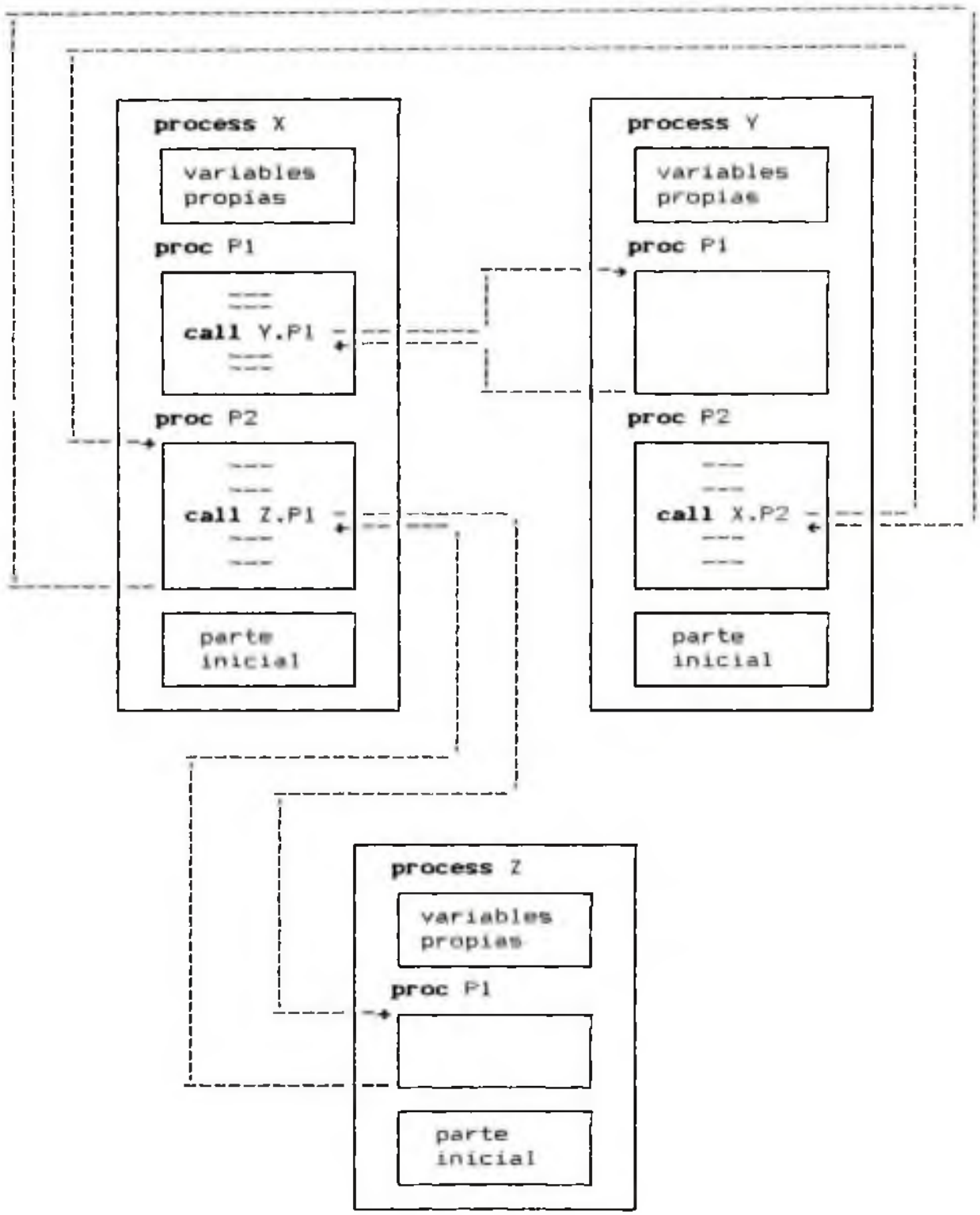


FIGURA 1.1 Esquema de un programa en el lenguaje procesos distribuídos de Hansen.

El no determinismo puede ser controlado por dos clases de porposiciones llamadas **comandos guardados y regiones guardadas**. Una región guardada puede retrazar una operación pero un comando guardado no.
Un comando guardado [Dijkstra, 1975],habilita un proceso a realizar una selección arbitraria de proposiciones inspeccionando el estado actual de sus variables. Sí ninguna de las alternativas es posible en el estado actual, el comando guardado no puede ser ejecutado y este puede ser omitido.
Los comandos guardados fueron definidos por E. W. Dijkstra (1975) y tienen la siguiente sintaxis y mecanismo.

if B1:S1 ¦ B2:S2 ¦ ... **end**
do B1:S1 ¦ B2:S2 ¦ ... **end**

Comando if.
Si alguna de las condiciones B1, B2, ..., es verdadera, entonces selecciona una de las condiciones verdaderas Bi y ejecuta la proposición Si que sigue a ésta; en otro caso termina el comando.
Comando do.
Mientras alguna de las condiciones B1, B2, ..., sea verdadera selecciona una de estas arbitrariamente y ejecuta la correspondiente proposición.

Las regiones guardadas hacen que un proceso espere hasta que el estado de las variables involucradas hagan posible un cambio arbitrario en la serie de proposiciones. Si ninguna de las alternativas es posible en el estado actual, el proceso pospone la ejecución de la región guardada.
Las regiones guardadas tienen la siguiente sintaxis y mecanismo.

when B1:S1 ¦ B2:S2 ¦ ... **end**
cycle B1:S1 ¦ B2:S2 ¦ ... **end**

Proposición when.

Espera hasta que una de las condiciones B1, B2, ... sea verdadera y ejecuta la correspondiente proposición.

Proposición cycle.

Ciclo permanente de la proposición when.

Si una serie de condiciones son verdaderas en una región o comando guardado, es impredecible cual de las correspondientes proposiciones seleccionará la máquina. Esta incertidumbre refleja el no determinismo natural de aplicaciones concurrentes.

Los tipos de datos usados en este lenguaje son:

int ... enteros

boll ... booleanos

char ... caracteres

set[n]T ... conjunto finito de tipo T

seq[n]T ... secuencia de n elementos de tipo T

array[n]T ... arreglo de n elementos de tipo T

Existe la proposición **for** que enumera todos los elementos en una estructura de datos

for x **in** y:S **end**

Proposición for.

Para cada elemento x en el conjunto o arreglo y, ejecuta la proposición S.

Una proposición for puede accesar y cambiar los valores de los elementos del arreglo, pero únicamente lee el valor de los elementos de un conjunto.

Por último, existe la proposición vacía denotada por **skip**.

A continuación se presentan dos ejemplos en este lenguaje.

Ejemplo 1. Buffer de mensajes.

Un proceso llamado buffer almacena una secuencia de caracteres que se transmiten entre procesos por medio de los

procedimientos send y receive.

```
process  buffer;
  S: seq[n] char
  proc send(c: char)
    when not S.full : S.put(c) end

  proc receive(#v: char)
    when not S.empty : S.get(v) end

  S := []
```

La primera operación que realiza el proceso es la parte inicial S := [], limpiar el buffer.
La parte inicial termina y el proceso realiza la segunda operación aceptando solicitudes externas. Otros procesos se pueden comunicar con el proceso buffer mediante las siguientes llamadas.

```
call  buffer.send(x)
call  buffer. receive(x)
```

Ejemplo 2. Asignador de un recurso al siguiente trabajo más corto.

En este ejemplo se presenta un asignador de un recurso en particular y concede el recurso al siguiente trabajo más corto entre n procesos de usuarios que hacen la petición.
Las solicitudes de servicio y el tiempo que usarán el recurso se implementan en una cola. El usuario espera hasta que la solicitud es seleccionada por el asignador.
Terminando el servicio se libera el recurso y este queda disponible.
El asignador espera hasta que una de dos situaciones se presenten:
1) Un proceso éntre o salga de la cola. El asignador buscará en la cola y seleccionará al siguiente usuario (en este

momento el asignador no concede el recurso).

2) Si el recurso está disponible y el siguiente usuario lo selecciona. El asignador concederá el recurso a este usuario y lo removerá de la cola.

Los procesos de los usuarios se identifican por un único indice 1, 2, ...,n. La constante **nil** denota un indice de proceso no definido.

El asignador usa las siguientes variables:

queue	Cola de indices de los procesos esperando
rank	Cola de tiempo que usarán el recurso los procesos que esperan.
user	índice de usuario que está usando el recurso
next	índice del siguiente usuario que usará el recurso

```
process  stc
   queue: set[n] int
   rank: array[n] int
   user, next, min: int

   proc request(who, time: int)
   begin
     queue.include(who)
     rank[who] := time
     next := nil
     when user = who : next := nil end
   end
   proc release;  user := nil
begin
  queue := []; user := nil; next := nil
  cycle
    not queue.empty & (next = nil) :
       min := maxinteger
       for i in queue
         if rank[i] > min : skip ;
            rank[i] <= min : next := i; min := rank[i]
```

```
          end
        end ;
    (user = nil) & (next <> nil) :
      user := next; queue.exclude(user)
  end
end
```

I.2 Especificaciones del diseño.

El lenguaje procesos distribuidos de Hansen es un lenguaje teórico y ha servido de inspiración para la creación de lenguajes distribuidos tales como: StarMod [Cook, 1980], Argus [Liskov y Sheifler, 1982] y Polux-s [Jan Janecek, 1989].
En este diseño también se toman ideas de este lenguaje debido a que está basado en llamadas a procedimientos remotos y pueden ser especificados como llamados a procedimientos externos en el diseño.
En el lenguaje procesos distribuidos cada proceso tiene sus propias variables y la comunicación entre procesos se realiza mediante llamados a procedimientos comunes, entonces un programa en este lenguaje se puede determinar de la siguiente manera:

Un programa que consiste de un número fijo de procesos concurrentes.

Cada proceso se ejecuta en una computadora. Esto indica la existencia de una red donde cada procesador está dedicado a un simple proceso y la comunicación entre procesos se realiza mediante llamados a procedimientos externos o remotos.

A este tipo de programa se le denominará un **programa distribuido.**

Para el diseño del lenguaje distribuido se toma como punto de partida al lenguaje concurrente Edison, siendo este un lenguaje implementado con regiones guardadas y procedual. Se diseñan las construcciones que hacen a este lenguaje un

lenguaje distribuido conservando la semántica de Edison, por tal razón a este nuevo lenguaje se le denominará en adelante, **Edison_D.**

El lenguaje Edison_D se le puede considerar un lenguaje de programación apropiado para la enseñanza de los principios de programación concurrente y para diseñar programas distribuidos.

Un proceso en Edison_D se define mediante: variables propias, módulos, procedimientos y parte inicial.

```
proc  nombre(parámetros)
  variables propias
  módulos
  procedimientos
  parte inicial
```

Un proceso en Edison_D es definido de manera similar que en los procesos distribuidos de Hansen. Faltaría explicar qué son los módulos en un proceso.

Un módulo es un cuerpo del proceso que contiene dos clases de entidades conocidas como: entidades locales y exportables, y realiza una operación inicial sobre estas entidades.

Las entidades locales se usan solamente en el módulo. Las entidades exportables pueden ser usadas en el módulo y fuera de éste.

Las entidades pueden ser declaraciones de tipos, variables y procedimientos.

La inicialización de un módulo tiene dos pasos:

1) Las entidades locales y exportables son creadas y adicionadas al contexto concurrente.
2) La operación inicial del módulo es realizada.

La principal aplicación de los módulos es para definir los monitores en Edison_D.

Un proceso en Edison_D realiza tres clases de operaciones: La primera es la inicialización de cada módulo declarado en el orden escrito, la segunda operación es el servicio a la

solicitud externa de otros procesos, y la tercera operación es realizar la operación inicial del proceso.
La primera operación se realiza de manera completa. Terminada ésta se realizan las dos operaciones restantes por interacción como en procesos distribuidos de Hansen.
A diferencia de los procesos distribuidos, el lenguaje Edison_D puede crear **procesos internos** dentro de cada proceso que forma un programa distribuido. Estos procesos internos aparecen y desaparecen al mismo tiempo mediante una proposición concurrente llamada **cobegin**, la cual se detalla en la implentación del lenguaje.

Para hacer distinción en los procesos, definiremos a un **proceso α** como un proceso definido en los procesos distribuidos de Hansen.
Un proceso interior es un proceso que se crea en un proceso α mediante la proposición cobegin.
También existen **procesos externos** en el lenguaje Edison_D, y corresponden a los servicios de solicitudes externas. Esto indica que para dar servicio a solicitudes externas, el sistema genera una serie de procesos externos que darán el servicio de llamar al procedimiento cuando se presente la solicitud externa.
En Edison_D existe declaraciones distribuidas (al inicio de cada proceso α), indicando qué procedimiento podrá ser llamado de otro proceso, ó qué procedimientos externos (en otros procesos) se pueden llamar.
Las declaraciones distribuidas indican que ciertos procedimientos del programa forman el procesamiento distribuido.
Por ejemplo, un proceso P puede llamar a un procedimiento R definido en otro proceso Q de la manera siguiente:

```
R (argumentos)
```

Previamente, existe la declaración distribuida del procedimiento R indicando que es externo y se localiza en el proceso Q. Además en el proceso Q existe la declaración distribuida

del procedimiento R indicando que es un procedimiento que podrá ser llamado por otros procesos.

Es evidente notar qué: para el proceso P la declaración distribuida es una **declaración externa** y para el proceso Q es una **declaración interna.**

Las construcciones para ambas declaraciones son definidas en el siguiente capítulo.

De acuerdo a las definiciones presentadas anteriormente, se define lo que significa un programa distribuido en el lenguaje Edison_D.

Un programa distribuido en Edison_D, consiste de un número fijo de procesos α que se ejecutan en un ambiente distribuido El ambiente distribuido indica que cada proceso α se ejecuta en un procesador con memoria independiente.

Un proceso α puede crear procesos internos y externos. El proceso interno se creará durante la ejecución de una proposición cobegin y desaparecen al termino de ésta.

Un proceso externo es un proceso permanente, se crea al inicio de ejecución del proceso α y responde a solicitudes externas de un procedimiento en particular.

A cada procedimiento que se solicite externamente, existe un proceso externo que dá servicio a la solicitud.

Un proceso interno es creado por el programador. Un proceso externo lo crea el lenguaje.

Las tres operaciones que realiza un proceso α se puede determinar como:

1) Inicialización de cada módulo declarado en el orden escrito.
2) Cada proceso externo dá servicio a solicitud externa
3) Operacion inicial

La primera operación se realiza de manera completa. Terminando esta operación, se realizan las dos operaciones restantes por interacción en procesos distribuidos.

II. IMPLEMENTACION

En este capítulo primeramente se describe las formas sintáticas del lenguaje Edison_D, posteriormente se menciona cada paso que realiza el compilador.
Un programa en Edison_D está compuesto por un número fijo de procesos α que se compilan por separado. Cada proceso α está compuesto por:

```
proc  nombre del proceso (parámetros)
   declaraciones distribuidas
   variables propias
   módulos
   procedimientos
   parte inicial
```

En las declaraciones distribuidas el compilador genera código Edison determinando parámetros relativos de los procedimientos externos que se invocan. A su vez, también genera direcciones relativas de los procedimientos internos que los invocan otros procesos.

El código Edison es interpretado por Kernel que implementa ciertas tablas para crear el ambiente distribuído. Estas tablas se crean de acuerdo a las declaraciones distribuidas especificadas en cada proceso α.

II.1 DESCRIPCION DEL LENGUAJE.

En el capítulo anterior se ha especificado que un programa distribuido en Edison_D consiste de un número fijo de procesos α que se ejecutan por separado en cada procesador con memoria independiente.
Para el lenguaje Edison_D, la manera de especificar que un proceso α se ejecutará en un procesador determinado significa que el proceso α se ejecutará en una estación de una red.
Actualmente el código que genera Edison_D se ejecuta en el procesador 8086-88, y se asigna a cada procesador como una estación en la red Edison.
La red Edison es pequeña de tipo bus que utiliza el método de pooling para asignar tiempo de servicio a cada estación. Existe una computadora que controla el acceso del bus determinando cierto tiempo de acceso a cada estación.

Para generar código ejecutable de un programa en un lenguaje distribuído existen dos maneras posibles:
La primera es especificar un programa con el número de procesos α que lo componen. El compilador generará el código para el programa, especificando la estación que correrá cada proceso. Así, en el momento de ejecución el programa asume la estación en que se encuentra y ejecuta solamente el proceso α para tal estación.
Como puede observarse, el código generado será el mismo para todas las estaciones, pero dependiendo de la estación se ejecutará el proceso α asignado.
La segunda manera de especificar un programa distribuido es mediante la obtención de código por separado para cada proceso α. El programador describe cada proceso α en un archivo independiente, así el compilador genera código por separado para cada proceso α. Cada proceso α es ejecutado en la estación correspondiente.
El código que se genera para cada estación es diferente ya

que corresponde a cada proceso α.
En Edison_D, el código que se genera para un programa distribuido se realiza mediante la segunda forma. Esto indica que cada proceso α del programa se especifica mediante un archivo fuente, el cual es entrada para el compilador y generará el código Edison para el proceso α en particular.
La figura 2.1 describe la compilación por separado de un programa distribuido.

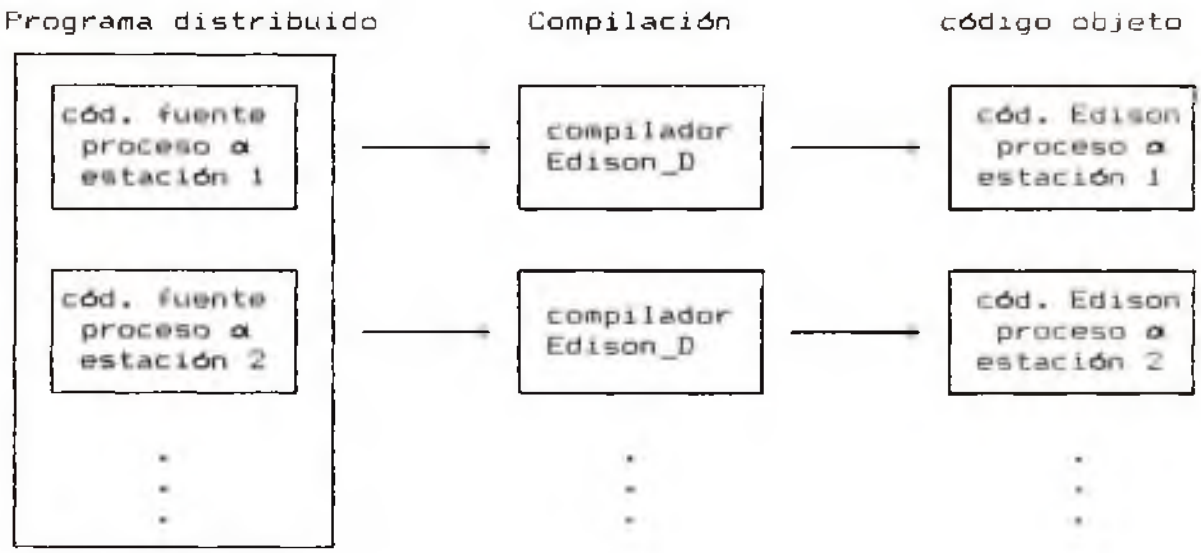


FIGURA 2.1 Compilación por separado de un programa distribuido.

Cada proceso α debe contener especificaciones básicas que lo hacen miembro o parte de un programa distribuido.
Las especificaciones básicas son declaraciones de entidades distribuidas. Las especificaciones básicas que se necesitan son las siguientes:

- El número de estación donde correrá el proceso α.
- Cuando se llame a un procedimiento en otro proceso α, se necesita especificar el número de estación donde se encuentra el proceso. Esto se logra declarando al procedimiento y la estación donde se encuentra.
- Por último un proceso α podrá responder a servicios de

solicitudes externas solo a ciertos procedimientos que sean previamente especificados.

Estas especificaciones forman el conjunto de declaraciones distribuidas en el lenguaje y posteriormente se describe las construcciones sintácticas.

El esquema general que se describe en la figura 2.2 representa a un proceso α compuesto de cuatro partes fundamentales.

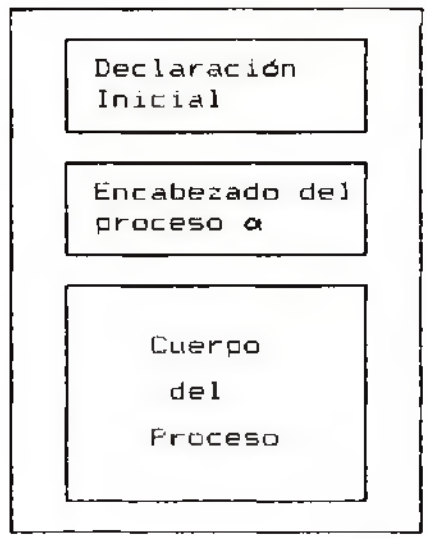


FIGURA 2.2 Esquema General de un proceso α

Declaracion Inicial: Son las declaraciones previas al encabezado del proceso. Estas declaraciones pueden ser declaraciones de constantes o declaraciones de tipo. Las declaraciones de constantes tienen la forma sintáctica.

<lista declaración constante> ::=
const <declaración constante> {; <declaración constante>}
<declaración constante> ::= <nombre> = <constante simbólica>

Las declaraciones de tipo son: **enum**, **record**, **array** y **set**.

<declaración tipo enum> ::=
enum <nombre> (<lista enumeración de símbolos>)
<declaración tipo record> ::=

record <nombre> (<lista de campos>)

<declaración tipo array> ::=

array <nombre> [<rango simbólico>] (<tipo>)

<declaración tipo set> ::= <nombre> (<tipo>)

Encabezado del Proceso: Nombre del proceso de una lista de parámetros opcional.

<encabezado proc> ::=

proc <nombre proc> ((<lista parametros>)) [: <tipo>]

Cuerpo del Proceso: Está definido mediante las siguientes partes.

Declaración Distribuida
Variables Propias
Módulos
Procedimientos
Parte Inicial

La declaración distribuida corresponde a tres construcciones en el lenguaje (**station, remote, entry**).

<declaración de estación> ::= **station** <número estación>

<declaración de procedimiento externo (remote)> ::=

remote <encabezado proc> **on** <número estación>

<declaración de procedimiento interno (entry)> ::=

entry <encabezado proc>

Declaración de estación: Bajo esta construcción se declara el número de estación que asignará el proceso α en el momento de ejecución.

Declaración de procedimiento externo (remote) : Bajo esta construcción se declara un procedimiento remoto (externo), definido en otra estación y al cual se le puede invocar.

Declaración de procedimiento interno (entry): Bajo esta construcción se declara un procedimiento entry (interno), definido en esta estación y podrá ser invocado por otra estación.

Para cada procedimiento declarado en entry, existirá un

proceso definido, proceso externo que se encargará de dar el servicio a una solicitud externa.
Estas construcciones forman las únicas declaraciones distribuidas.

Anteriormente se há mencionado que el lenguaje Edison_D realiza tres clases de operaciones.

1) Inicialización de los módulos en el orden escrito
2) Solicitud externa
3) Parte inicial

Precisamente, la operación 2 (solicitud externa), es una operación que se realiza bajo los procedimientos declarados en **entry**. Esto indica que sólo los procedimientos declarados **entry** tendrán servicio de un llamado externo.
Con las tres construcciones se realiza la declaración distribuida en el proceso α. Téngase presente que en esta parte sólo se declara al procedimiento (<encabezado proc>) y no a su cuerpo. Por ejemplo en la declaración **entry** sólo se declara al procedimiento y posteriormente en el cuerpo del proceso α se declara el procedimiento completo.

Las variables propias declaran variables que solo se usan en el proceso. De hecho no existe variables comunes en el lenguaje. En esta parte pueden existir declaraciones como en la declaración inicial. Las variables que se declaran corresponden a variables de tipo estardard o de tipo declarado previamente.

<lista declaración variables> ::=
    var <grupo variables> { ; <grupo variables> }
<grupo variable> ::=
    <nombre variable> { , <nombre variable> } : (tipo)
<nombre variable> ::= <nombre>

Los módulos en un proceso se definen de la siguiente manera: Un módulo es un cuerpo del proceso que contiene dos clases de entidades conocidas como entidades locales y exportables, y

realiza una operación inicial sobre estas entidades. Las entidades locales solamente se pueden accesar dentro del módulo, pero las entidades exportables pueden usarse dentro y fuera del módulo a partir de la definición.
La sintaxis del módulo es:

```
<declaración módulo> ::=
   { <declaración local> | <declaración exportable> }
   <parte inicial>

<declaración local> ::= <declaración>
<declaración exportable> ::= * <declaración>
<parte inicial> ::= begin <lista proposiciones> end
<declaración> ::= <lista declaración constante> |
                  <declaración de tipos> |
                  <lista declaración variables> |
                  <declaración procedimiento>
```

De acuerdo a la definición, un módulo puede contener constantes, declaración de tipos, variables, procedimientos que pueden ser locales y exportables, además de la parte inicial que es ejecutada en la primera operación que realiza el proceso α.

Los procedimientos se dividen en tres clases: procedimiento completo, pre-post definición de procedimiento y procedimiento de librería.
Un procedimiento completo tiene su encabezado y cuerpo.
El cuerpo del procedimiento está compuesto por una parte de declaración de constantes de tipos, variables y la parte inicial.
La pre-definición del procedimiento se utiliza cuando en el proceso se realizan procedimientos mutuamente recursivos.
Cuando un procedimiento llama a un procedimiento aun no definido se realiza una pre-definición, de otra forma el compilador marca error al llamado. La sintaxis es:

<pre declaración procedimiento> ::= **pre** <encabezado proc>

Cuando más adelante se defina el procedimiento, es necesario realizar una post definición que indica al compilador que el procedimiento se definió mediante la predefinición. La sintaxis es:

<post declaración procedimiento> ::=
post <procedimiento completo>

Por último, la parte inicial tiene la sintaxis.

<parte inicial> ::= **begin** <lista de proposiciones> **end**
<lista de proposiciones> ::= <proposición> {; <proposición>}
<proposición> ::= **skip** | <variable simbolo> := <expresión> |
<llamado a procedimiento> |
if <lista de proposición condicional> **end** |
while <lista de proposición condicional> **end** |
when <lista de proposición condicional> **end** |
cobegin <lista de procesos> **end**

II.2 GENERACION DE CODIGO.

El compilador Edison_D está dividido en cuatro pasos. Cada paso realiza una exploración al texto fuente y genera código intermedio a disco.

Paso 1. Análisis Léxico.

Explora el programa fuente (texto), caracter por caracter y convierte los símbolos a valores enteros o token's.
Los token's de los símbolos es depositado en el archivo temp1. Este paso no distingue entre diferentes usos del mismo nombre en diferentes bloques.

Paso 2. Análisis de Sintaxis y Alcance.
entrada: temp1
salida: temp2

Se checa la salida del pasol (archivo temp1), usando descendencia recursiva con un procedimiento por cada forma sintáctica del lenguaje. Diferentes usos del mismo símbolo en diferentes bloques son reemplazados por un único índice del nombre. Aparte, analiza el alcance que tiene el nombre al ser usado en el contexto, pero no analiza a que clase de entidad hace referencia.

Paso 3. Análisis Semántico.
entrada: temp2
salida: temp1

Se compara qué operandos y operadores son compatibles, calcula la longitud de tipos y las direcciones de las variables. Nuevamente se utiliza el método descendente recursivo como en el paso 2.

Paso 4. Generación de Código.
entrada:temp1
salida: temp2

En este paso se construye una tabla de referencias y un stack de procedimientos, se explora si la salida del código abstracto en la cual las referencia son reemplazados por direcciones relativas.
El código generado es un código llamado código Edison el cual es interpretado por el Kernel.
El código Edison está compuesto por primitivas y sus correspondientes argumentos. El compilador Edison_D genera 75 primitivas que serán interpretadas por el Kernel. Estas primitivas son las siguientes.

Primitivas de Código estandar.
add4, also4, and4, assign4, blank4, cobegin4, constant4, construct4, difference4, divide4, do4, else4, endcode4, endlib4, endproc4, endwhen4, equal4, field4, goto4, greater4, in4, index4, instance4, intersection4, less4, libproc4, minus4, modulo4, multiply4, newline4, not4, notequal4, notgreater4, notless4, or4, paramarg4, paramcall4, procarg4, proccal4, procedure4, process4, subtract4, union4, valspace4, value4, variable4, wait4, when4, addr4, halt4, obtain4, place4, sense4, elemassign4, elemvalue, localcase4, localset4, localvalue4, localvar4, outercall4, outercase4, outerparam4, outerset4, outervalue4, outervar4, setconst4, singleton4, stringconst4,

Primitivas de tipo distribuido.
@defentry4, @entry4, @execute4, @initial4, @remote4, @remotecall4, @station4.

Cada primitiva tiene un número asignado entre 0 - 74 en el orden escrito: add4 = 0, also4 = 1, and4 = 2, assign4 = 3, ... @remote4 = 72, @remotecall4 = 73, @station = 74.

A cada primitiva el compilador genera el código:
1792 + 3 * (número de primitiva)

Por ejemplo:

Primitivas	código Edison		
add4	1792 + 3 * 0	=	1792
also4	1792 + 3 * 1	=	1795
and4	1792 + 3 * 2	=	1798
assign4	1792 + 3 * 3	=	1801
.	.		.
.	.		.
.	.		.
@remote4	1792 + 3 * 72	=	2008
@remotecall4	1792 + 3 * 73	=	2011
@station4	1792 + 3 * 74	=	2014

Cada primitiva puede contener parámetros o datos que indican saltos relativos, constantes, direcciones, número de líneas, y otros.
El código Edison se puede considerar como un subconjunto ordenado de enteros de la siguiente manera:

Código Edison = secuencia de primitivas + secuencia de datos
= { 1792, 1795, 1798, ..., 2011, 2014 } +
{ x | -32768 ≤ x ≤ 32767 }

Cada primitiva es interpretada por el Kernel mediante un vector que atiende a cada primitiva. Este vector llamado **opcode** comienza a partir de la dirección física 1792 y termina en la dirección 2017. El kernel interpreta cada código mediante un salto a la rutina que dá el servicio. Por ejemplo para la primitiva add4 = 1792, existe la instrucción jump (**addx**) en las direcciones 1792, 1793, 1794. La rutina **addx** atiende a la primitiva add4. Para la primitiva also4 = 1795, existe la instrucción jump (**alsox**) en las direcciones 1795, 1796, 1797.
El vector de interpretación es la rutina **opcode** con la siguiente información:

```
pad  1792
do opcode:
   jump (addx)
   jump (alsox)
   jump (andx)
   jump (assign)
        .
        .
        .
   jump (remote)
   jump (remotecall)
   jump (station)
```

Las primitivas de tipo distribuido crean el ambiente necesario para llamados a procedimientos remotos. Estas primitivas son derivadas de las tres construcciones distribuidas **(station, remote, entry)**. Estas primitivas se describen a continuación y se estudian (implantación) en el siguiente capítulo.

Primitiva **@initial**: Es la primer primitiva en todo proceso α. Se encarga de inicializar la Tabla de Procedimientos Distribuidos, la Tabla de solicitudes, el número de procesos externos y un apuntador a la Cola de Estados de los Procesos.

Tabla de Procedimientos Distribuidos: Es una tabla con descripción de todos los procedimientos distribuidos del proceso. Cada proceso α tiene una Tabla de Procedimientos Distribuidos. Todas son idénticas sólo cambia en la declaración (Para un proceso α, algunos procedimientos son remote mientras otros son entry). La tabla contiene: tipo de procedimiento, longitud de regreso (caso función) dirección absoluta (caso entry), número de parámetros, cada tipo y longitud del valor de cada parámetro. Una descripción total de la tabla se menciona en el siguiente

capítulo.

Tabla de solicitudes: Tabla con solicitudes externas de cada procedimiento.

Cada posición de la tabla corresponde a una solicitud de procedimiento.

Sí solicitud [i] = false ... No hay solicitud para el procedimiento i.

Sí solicitud [i] = true ... Existe solicitud para el procedimiento i.

Primitiva **@station:** Define la estación en el momento de corrida del proceso α.

Posiblemente antes de correr el proceso α, la computadora tenga asignada un número de estación. Cuando el proceso α se ejecuta, la primitiva @station cambia la asignación previa por el número de estación dispuesto en el proceso α.

Primitiva **@remote:** Define un procedimiento externo en la estación descrita.

Primitiva **@entry:** Define un procedimiento interno que podrá ser solicitado externamente.

Primitiva **@execute:** Se encarga de activar las operaciones 2 y 3 de todo proceso α.

Primitiva **@remotecall:** Prepara los argumentos para el llamado a un procedimiento remoto.

Primitiva **@defentry:** Define un procedimiento entry. Esta es similar a la primitiva **procedure.**

El compilador Edison_D fué como resultado de anexar al lenguaje Edison construcciones **station, remote** y **entry.** definidas en la primera parte de este capítulo. Tal implementación se realizó en cada paso del compilador Edison,

resultando las primitivas de tipo distribuido.
El funcionamiento de cada primitiva distribuida se explica en el siguiente capítulo donde se implementa el RPC. El resto de este capítulo se presenta un ejemplo de un proceso en el cual se muestra el código de cada paso del compilador.

Ejemplo:

El proceso que se presenta es llamado "producer" y es un proceso α en un programa distribuido compuesto de tres procesos α. Estos tres procesos son "producer", "buffer" y "consumer".
El proceso "producer" se encarga de leer nombres de la terminal y enviarlo al proceso "buffer" que se encarga de almacenarlo en un arreglo de nombres. Otro proceso llamado "consumer" se encarga de pedir un nombre del arreglo y escribir el nombre en la terminal.

```
const namelength = 12 "characters"
array name [1:namelength] (char)

proc producer(
   proc display(value: char);
   proc accept(val value: char);
   proc readname(proc read(var c: char); var value: name);
   proc writename(proc write(c: char); value: name)
         )

station 1
remote proc deposita (mensaje: name) on 2
remote proc envia (var mensaje: name) on 2

var nombre: name; band: bool

begin
  band := true;
```

```
  while band do
    readname(accept, nombre);
    deposita(nombre)
  end
end
```

Como se puede observar, el proceso inicia con la declaración inicial de **const** y **array**. Luego define el proceso α con el nombre de producer. Este proceso tiene cuatro procedimientos como parámetros que estan definido en el S. O.
La construcción **station** 1, declara que en el momento de ejecución la estación donde corra este proceso asumirá el número 1 como estación (terminado el proceso, la estación toma su número original).
La primera construcción **remote** declara como remoto o externo al procedimiento deposita con un parametro por valor y está sobre la estación 2.
La segunda construcción **remote** declara como remoto o externo al procedimiento envia con un parametro por dirección y está sobre la estación 2.
El parámetro de este procedimiento se debe entender como un parámetro por dirección parecido al de por variable, aunque la sintaxis indique que son iguales.
Debe mencionarse que el orden de las declaraciones debe conservarse en los procesos "buffer" y "consumer" ya que el mecanismo de RPC es sobre el número de procedimiento que aparece en las declaraciones distribuidas.
También se debe a que la compilación se realiza por separado a cada proceso α y este mecanismo es fácil de implementar.
La parte inicial es un ciclo permanente que se encarga de leer un nombre en la terminal y llamar a procedimiento deposita definido en la estación 2 que se encarga de meterlo en un buffer de nombres.
El código intermedio producido por el paso 3 representa ciertas etiquetas de cada construcción. Las etiquetas permiten determinar direcciones relativas en la generación del código Edison del paso 4.

El código intermedio del paso 3 y el código generado por el paso 4 se presentan las siguientes hojas.

```
ARCHIVO: productext
Código intermedio
Compilador Edison-PC Distribuido
paso 3


newline2     1
newline2     2
newline2     3
newline2     4
newline2     5
newline2     6
newline2     7
newline2     8
newline2     9
newline2    10
newline2    11   initial2  procedure2   1    8    9    10
                 station2   1
newline2    12
newline2    13   remote2    1    0    1    1    12
newline2    14
newline2    15   remote2    2    0    1    1    12
newline2    16
newline2    17
newline2    18
newline2    19
newline2    20    execute2   variable2   1    12   constant2   1
                  assign2    1
newline2    21    while2    11    variable2    1    12
                  value2     1
newline2    22    do2    12    paramarg2    1    2
                  variable2    1    0    paramcall2    1    4    3
newline2    23    variable2    1    0    value2    12
newline2    24    remotecall2    1    12    2    else2    11    12
newline2    25
newline2    26
newline2    27    endproc2    9    13    10    8
                  endcode2
```

ARCHIVO: productext
Código Generado
Compilador Edison-PC Distribuido
paso 4

Dir Relativa	Código Generado	
[0]	108	
[2]	1792 + 3 * 71	@initial4
[4]	1792 + 3 * 39	@procedure4
[6]	16	
[8]	26	
[10]	24	
[12]	11	
[14]	1792 + 3 * 74	@station4
[16]	1	
[18]	1792 + 3 * 72	@remote4
[20]	1	
[22]	0	
[24]	1	
[26]	1	
[28]	24	
[30]	1792 + 3 * 72	@remote4
[32]	2	
[34]	0	
[36]	1	
[38]	2	
[40]	24	
[42]	1792 + 3 * 70	@execute4
[44]	1792 + 3 * 58	@localvar4
[46]	34	
[48]	1792 + 3 * 6	@constant4
[50]	1	
[52]	1792 + 3 * 53	@elemassign4
[54]	1792 + 3 * 57	@localvalue4
[56]	34	
[58]	1792 + 3 * 10	@do4
[60]	44	
[62]	1792 + 3 * 22	@instance4
[64]	0	
[66]	1792 + 3 * 35	@paramarg4
[68]	-12	
[70]	1792 + 3 * 58	@localvar4
[72]	10	
[74]	1792 + 3 * 22	@instance4
[76]	0	
[78]	1792 + 3 * 36	@paramcall4
[80]	-8	
[82]	1792 + 3 * 58	@localvar4

```
[ 84]          10
[ 86]        1792 + 3 * 44          @value4
[ 88]          24
[ 90]        1792 + 3 * 73          @remotecall4
[ 92]          1
[ 94]          24
[ 96]          2
[ 98]        1792 + 3 * 11          @else4
[100]         -44
[102]        1792 + 3 * 14          @endproc4
[104]        1792 + 3 * 12          @encode4
[106]          26
```

El listado del código generado (Edison) está compuesto de 3 columnas. La primera columna indica la dirección relativa del código (2 bytes por código). La segunda columna representa el código (puede ser una primitiva ó un dato), en el caso de una primitiva la tercera columna indica el tipo.

La dirección 0 contiene el número de bytes del código generado. La dirección 2 es la primitiva @initial4 que realiza la inicialización de las tablas de uso distribuido. La dirección 4 es la primitiva @procedure4, indica el proceso α con los datos (parámetros) en las direcciones 6, 8, 10 y 12 que especifican:

dirección 6	16 bytes para parámetros de entrada
dirección 8	26 bytes para variables locales en el proceso
dirección 10	24 bytes para variables temporales
dirección 12	número de línea de programa fuente. Se usa para denotar error en la línea 11 en caso de no existir memoria disponible para variables locales o temporales.

La dirección 14 contiene la primitiva @station4 que declara la estación 1 (dirección 16) en el momento de correr el proceso.
La dirección 18 contiene la primitiva @remote4 que contiene 5 parámetros (direcciones 20, 22, 24, 26, 28).

dirección 20	número de procedimiento distribuido
dirección 22	longitud en bytes de regreso

dirección 24 número de parámetros del procedimiento
dirección 26 tipo variable (1 indica por valor)
dirección 28 longitud en bytes del argumento

La dirección 30 es similar (primitiva @remote4).
La dirección 42 contiene la primitiva @execute4, indica la ejecución de las operaciones 2 y 3 del proceso.

La dirección 90 contiene la primitiva @remotecall4 que contiene 3 parámetros direcciones (92, 94, 96).

dirección 92 llamado al procedimiento 1
dirección 94 longitud de argumentos igual a 24 bytes
dirección 96 El procedimiento solicitado se encuentra en la estación 2.

La primitiva @remotecall4 crea el llamado al procedimiento remoto. La implementación se describe en el siguiente capítulo.

Por último se menciona la dirección 104 (primitiva @encode4) que contiene el parámetro número de línea (igual a 26). Con esta primitiva termina la operación 3 (parte inicial) del proceso y continua solamente la operación 2 (solicitudes externas) del proceso.

III. LLAMADO A PROCEDIMIENTOS REMOTOS.

Cuando el compilador Edison_D ha generado código para un proceso α, este código llamado código Edison será interpretado por el Kernel en el momento de ejecución del proceso. El compilador Edison_D genera las primitivas necesarias para realizar llamadas a los procedimientos remotos. Estas primitivas son interpretadas por el Kernel realizando el mecanismo de comunicar dos procesos.
La figura 3.1 muestra los componentes necesarios usados para implementar RPC.

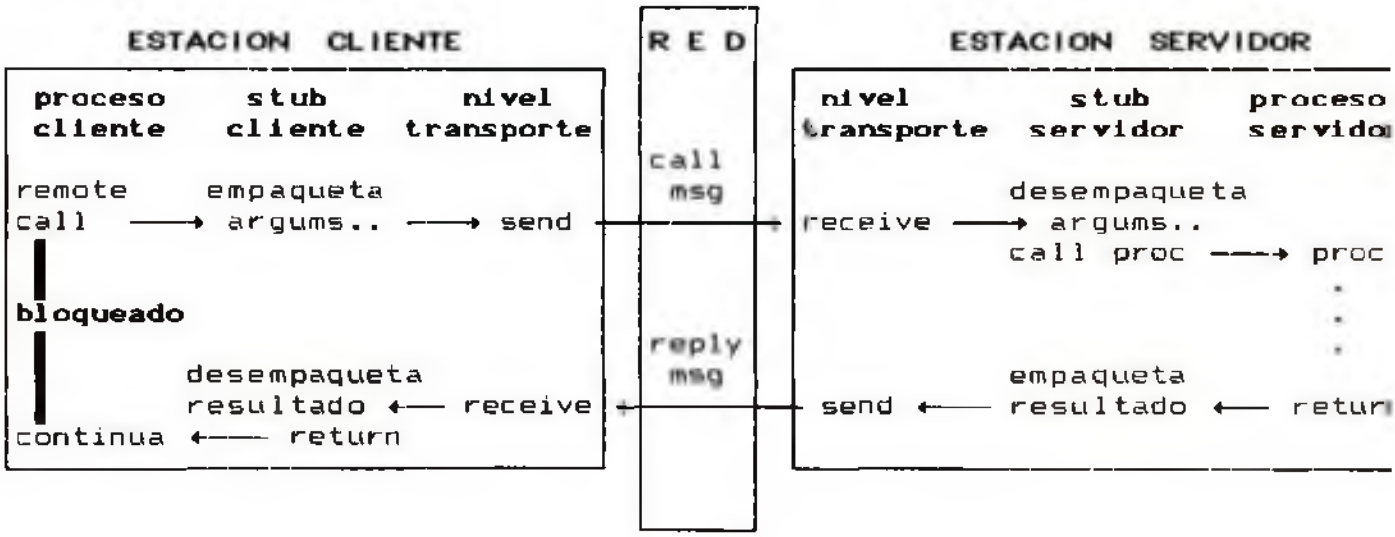


FIGURA 3.1 Componentes para implementación de RPC

La figura describe a dos estaciones cliente y servidor en una red. Durante la ejecución del proceso en la estación cliente, se realiza un llamado a un procedimiento remoto (remotecall), para tal efecto existe un procedimiento llamado Stub que se encarga de empaquetar los argumentos y ponerlos en el paquete para ser transmitido (por medio de mensaje) a la red mediante

una operación **send**. A partir de ese instante, el proceso cliente queda bloqueado esperando respuesta por el Stub.
Cuando el mensaje llega a la estación servidor mediante la operación **receive**, se recibe el paquete, se llama al Stub servidor que se encarga de desempaquetar los argumentos. El Stub llama al procedimiento solicitado, y se realiza la ejecución del procedimiento. Cuando el procedimiento termina, el control pasa al Stub que se encarga de empaquetar los resultados y solicita a la red el envío del paquete (**send**).
El paquete es transmitido en la red por medio de mensaje a la estación cliente, esta la recibe mediante la operación **receive**. El Stub cliente se encarga de desempaquetar el resultado y ponerlo en las direcciones adecuadas, luego desbloquea el proceso.
El proceso continúa con la ejecución de la instrucción siguiente a **remotecall**.

III.1 IMPLEMENTACION DE PRIMITIVAS.

Este mecanismo RPC se ha implementado en el Sistema Edison de la siguiente manera: El compilador Edison_D genera código del proceso, este código es interpretado por el Kernel. En el código generado existen primitivas de tipo distribuidas que al ser interpretadas por el Kernel darán un ambiente distribuido.
Las primitivas mencionadas son las siguientes:

@station	...	define estación
@remote	...	declara procedimiento remote
@entry	...	declara procedimiento interno
@defentry	...	define procedimiento entry
@execute	...	inicia operación 2 y 3 del proceso
@remotecall	...	llamado a procedimiento remoto

Estas primitivas son interpretadas por el Kernel para realizar el ambiente distribuido. Cada primitiva y sus argumentos se explican a continuación.

PRIMITIVA. **@station**
argumento: station process

La primitiva @station solicita a la red que mientras se ejecute el proceso, estará asignado bajo el número de estación station process.
Cuando el proceso termina se libera el número de estación y la estación continúa con el número asignado previamente.
La rutina que lleva a cabo la primitiva @station es una rutina pequeña.

```
do  station:
    move (st[p+2], ax)
    move (ax, st[MyAddr])
    add (4, p)
    jumpx (st[p])
```

La primera instrucción mueve el contenido de la dirección p+2 al registro ax. Este contenido es el argumento station process.
La segunda instrucción mueve el contenido del registro ax, al contenido de la dirección MyAddr. La dirección MyAddr le sirve a la red para representar el número de estación.
Las dos últimas instrucciones realizan un salto a la siguiente instrucción del proceso. Recuerde que el registro p contiene la próxima instrucción a ejecutar.

PRIMITIVA. **@remote**

argumentos: num proc ... número de procedimiento distribuido.

log reg ... longitud en bytes del valor que regresa el procedimiento.

nparam ... número de parametros que contiene el procedimiento (Máximo 10).

(tipo ... tipo del primer parámetro

long ... longitud en bytes del tipo

.

.

.) Estos argumentos solo existen si el nparam es mayor que cero.

La primitiva @remote inserta el procedimiento declarado por num proc en la Tabla de procedimientos distribuidos.
La tabla de procedimientos distribuidos contiene la información necesaria de todos los procedimientos distribuidos del programa.

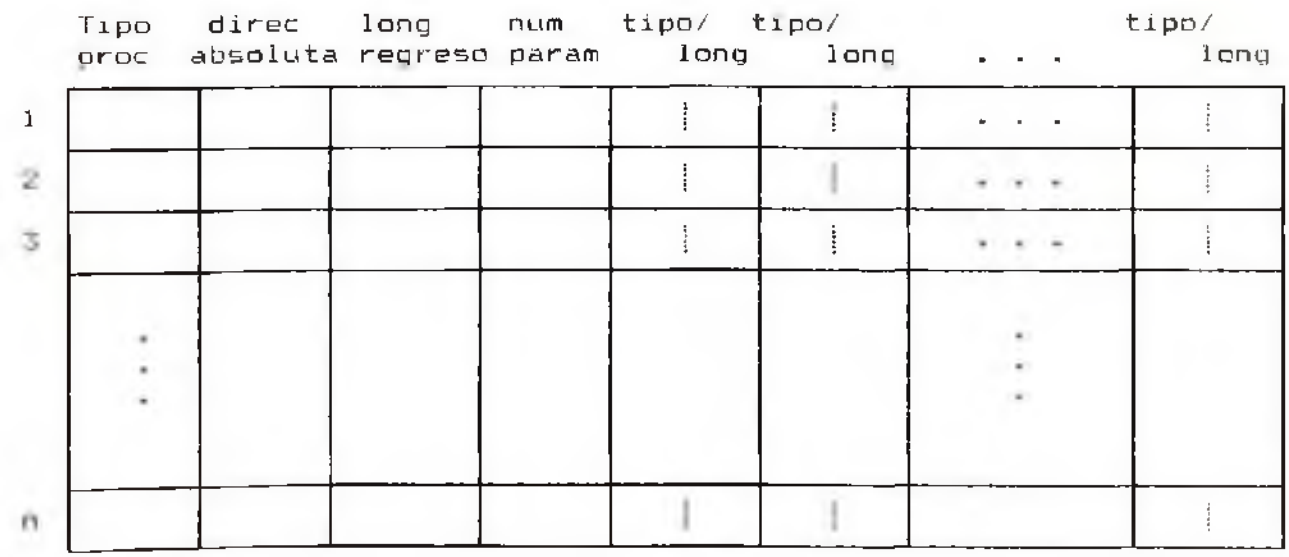


FIGURA 3.2 Tabla de Procedimientos Distribuidos

La Tabla de procedimientos distribuidos contiene la información de cada procedimiento en un programa distribuido. Esta tabla se encuentra en cada proceso α que compone el programa. Esto indica que en cada estación se encuentra la misma tabla con una pequeña diferencia en el campo Tipo proc (tipo procedimiento), debido que el para algunas estaciones un procedimiento será Tipo proc = **remote**, pero para la estación donde se define el procedimiento, el campo será Tipo proc = **entry**. Cada campo de la tabla indica lo siguiente.

Tipo proc ... **remote** | **entry**

direc absoluta ... Es la dirección donde inicia el procedimiento en el momento de ejecución del proceso. Si Tipo proc = **remote**, la dirección no tiene sentido (Tipo proc = 0).

long regreso ... Es la longitud (en bytes), del valor que regresa el procedimiento remoto.

num param ... Número de parámetros que contiene el procedimiento. Máximo 10.

Tipo ... valor | dirección

long ... Longitud (en bytes), del tamaño del tipo de variable.

La tabla de procedimientos distribuídos puede aceptar como máximo n procedimientos. Actualmente el valor de n es 10. Cada posición o número de procedimiento distribuido es único y es asignado por el compilador Edison_D.

PRIMITIVA. **@entry**

argumentos: desp ... desplazamiento relativo al procedimiento.

num proc ... número de procedimiento distribuido.

log reg ... longitud en bytes del valor que regresa el procedimiento.

nparam ... número de parametros que contiene el procedimiento (Máximo 10).

{ tipo ... tipo del primer parámetros

long ... longitud en bytes del tipo

.

.

. } Estos argumentos solo existen si el nparam es mayor que cero.

La primitiva @entry define a un procedimiento el cual prodrá ser llamado externamente. Esta primitiva crea un proceso externo el cual responderá a la solicitud externa. Este proceso externo se mantiene bloqueado pero es habilitado cuando se realiza la operación 2 del proceso α.

1) Inserta el procedimiento declarado por num proc en la Tabla de procedimientos distribuidos.
La tabla de procedimientos distribuidos contiene la información necesaria de todos los procedimientos distribuidos del programa.

2) Crea un area o stack de variable para el procedimiento entry.

3) Obtiene el estado de los registros para un proceso externo que es creado para responder a las solicitudes del procedimiento entry. El estado los deposita en la cola de estados de procesos.

La primera operación que realiza la primitiva @entry es similar a la que realiza @remote. La diferencia radica en que el campo dirección absoluta de la Tabla de procedimientos distribuidos contiene la dirección de inicio del procedimiento. Esta dirección se calcula

direcc absoluta = p + st[p+2]

El contenido de la dirección p+2 es el argumento desp (desplazamiento relativo).

La operación 2 de la primitiva @entry se describe mediante la siguiente figura.

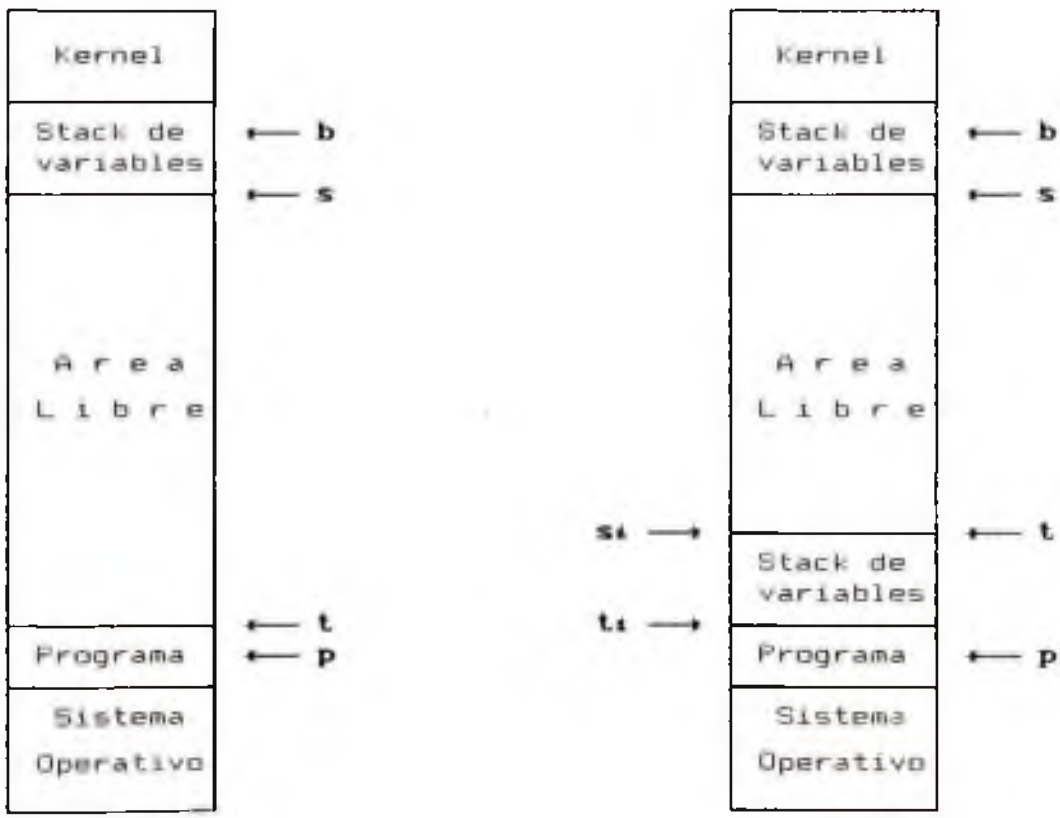
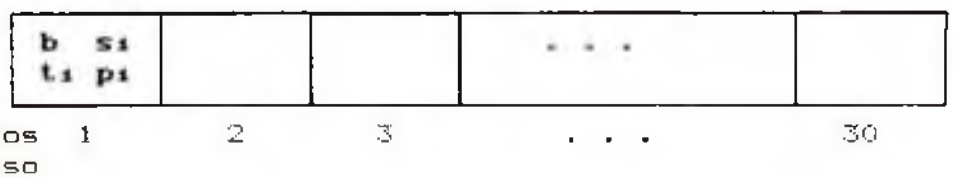


FIGURA 3.3 Operación 2 de la primitiva **@entry**

La operación 2 de la primitiva @entry crea un stack de variables para el procedimiento entry.
Como se puede ver en la figura 3.3 (izquierda), el proceso α representado como un módulo de programa es almacenado por el Sistema Operativo (durante la ejecución), en la parte inmediata inferior de este último.
En la operación 2, figura 3.3 (derecha) el stack de variables para el procedimiento entry se crea en la parte inmediata inferior del registro **t**. En este stack se pondrá la instancia de llamado cuando el proceso externo reciba una solicitud externa de llamado al procedimiento entry.

La operación 3, desposita los estados de los regitros **b**, s_1, t_1 y p_1 en la Cola de Estados de los Procesos.

Cola de Estados de los Procesos



Sí el lector se ha dado cuenta, en ambas figuras de las operaciones 2 y 3, se presenta la primitiva @entry como la primera que el proceso α encuentra. Sí existieran más primitivas @entry realizaría el mecanismo similar.
Para la operación 3, se anexa el nuevo estado de los registros en la siguiente posición disponible de la cola.
Solo falta mencionar. El registro p_1 apunta al inicio del proceso externo para solicitud del procedimiento con número de procedimiento distribuido igual a 1.

PRIMITIVA. **@defentry**

argumentos:	paramlength	...	longitud (en bytes) de los parámetros
	varlabel	...	longitud (en bytes) que ocupan las variables locales al procedimiento.
	templabel	...	longitud que ocupan las variables de uso temporal.
	nproc	...	número de procedimiento
	lineno	...	número de línea

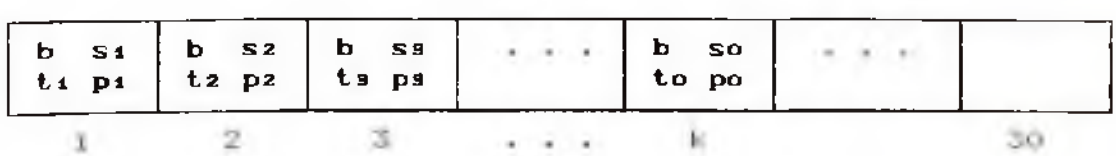
La primitiva @defentry es similar a @procedure. Es de ayuda para determinar el desplazamiento desde una primitiva @entry.

PRIMITIVA. **@execute**

Esta primitiva no contiene argumentos. La función de esta primitiva es la de iniciar con las operaciones 2 y 3 de un proceso α. La manera que realiza las operaciones es:

1) Depositar los estados de los registros del proceso α en la Cola de Estados de los Procesos.
2) Crear un ciclo permanente de la operación **cobegin** con los procesos externo y el proceso α.

Se podría entender que la Cola de Estados de los Procesos está compuesta de la siguiente forma:



Los primeros k-1 estados corresponden a los estados de k-1 procesos externos que fueron creados al ser declarados cada procedimiento entry. El estado k de los registros, corresponde a los registros del proceso α. De esta forma al comenzar la parte inicial del proceso α se tiene en la cola los estados de cada proceso externo y por último el estado de registros del proceso α.

Se puede entender que la operaciones 2 y 3 del proceso α, corresponde a realizar un ciclo permanente de la operación **cobegin** de los k procesos.

```
cobegin 1 do proceso 1
   also 2 do proceso 2
   also 3 do proceso 3
            .
            .
            .
   also k do proceso k
end
```

Cuando se ejecuta el proceso k (parte inicial del proceso α), puede existir una operación **cobegin** para procesos creados por el programador (llamado procesos internos), entonces en la cola se anexan estos nuevos procesos y comparten el procesador sólo los procesos internos. Cuando la operación **cobegin** termina, los procesos internos desaparecen y nuevamente comparten el procesador los k procesos (k-1 procesos externos y el proceso α).
De esta manera el Sistema Edison realiza las operaciones 2 y 3 del proceso α.

PRIMITIVA. **@remotecall**

argumentos:	num proc	...	número de procedimiento distribuído.
	long	...	longitud de los valores enviados
	nstation	...	número de estación donde se encuentra el procedimiento.

La primitiva @remotecall realiza primero la información necesaria para que el procedimiento Stub_Out realice el llamado al procedimiento. Esta información la presenta en los campos:

Función = 1		Indica que se requiere el servicio remotecall
num proc		número del rpocedimiento distribuido
long args		longitud (bytes) de los argumentos. Es importante para informar a la red la longitud del total de los argumentos.
station out		Estación donde se encuentra el procedimiento.

Después de generar la información, la primitiva @remotecall llama a Stub Out (mientras permanece bloqueado hasta esperar respuesta).
El procedimiento Stub Out con la información y la Tabla de Procedimientos Distribuidos se encarga de empaquetar los argumentos y crear el encabezado del paquete.
La Tabla de Procedimientos Distribuidos le indica al Stub Out el tipo de cada argumento y la longitud (bytes) del argumentos.
La figura 3.4 presenta el esquema del empaquetamiento de argumentos y control del encabezado del paquete.

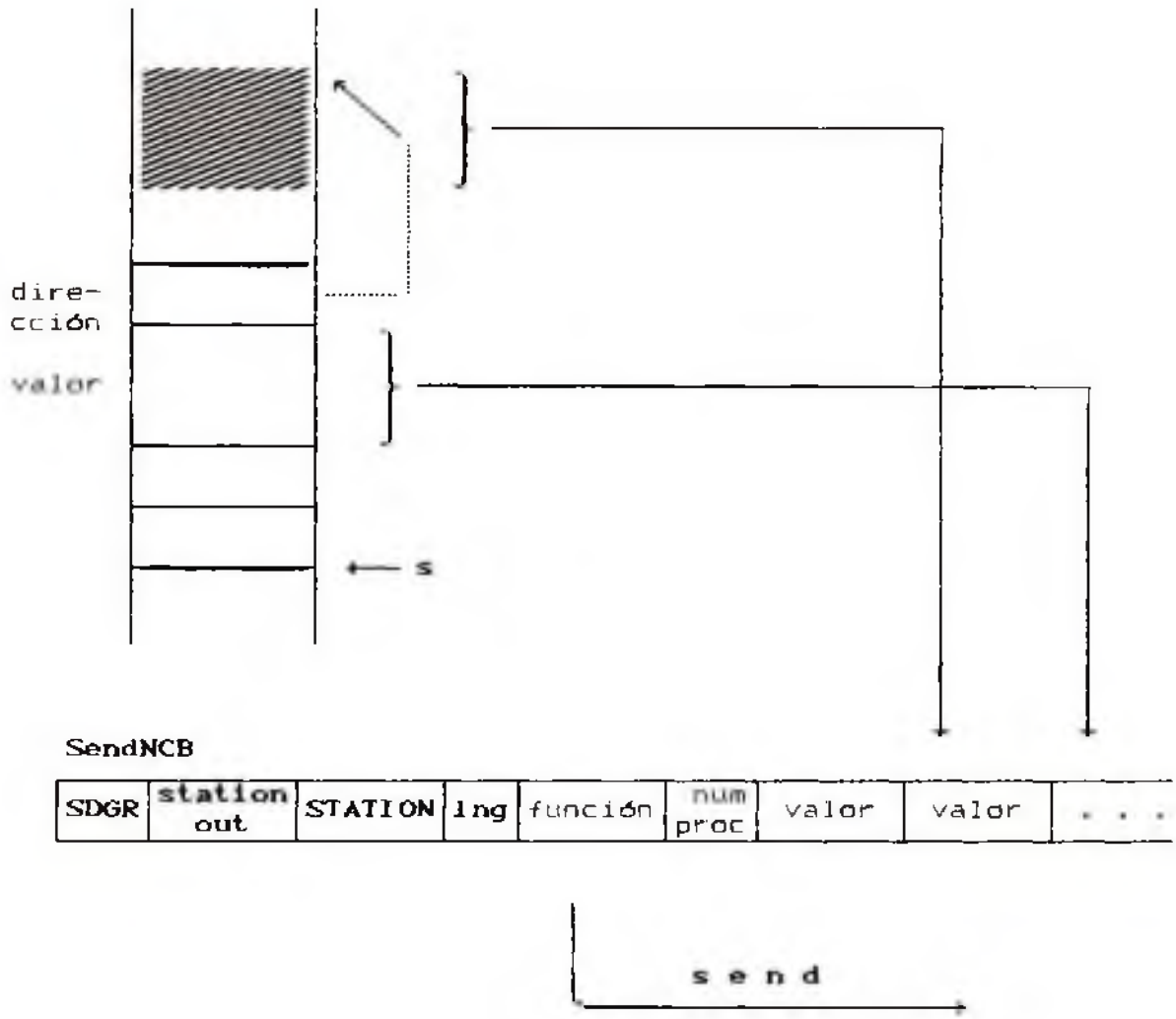


FIGURA 3.4 Esquema del funcionamiento de Stub Out con función 1.

Previamente al llamado **remotecall** se crea la instancia de llamado en el stack de variables del proceso activo.
El Stub Out desempaqueta los argumentos de acuerdo a la información en la Tabla de Procedimientos Distribuidos. Si es tipo valor se transfiere la longitud de bytes descrita en la tabla. Si el argumento es de tipo dirección se transfiere la zona de memoria donde apunta el contenido del argumento.

El control de encabezado del paquete contiene la información:

SDGR ... Solicitud a la red del comando Send Datagrama
station out ... Estación destino o receptora
STATION ... Estación fuente o transmisora
lng ... Longitud de bytes que tiene el mensaje

El Stub Out llama a la red MILAN que se encarga de enviar **(send)** el mensaje a la estación destino. La respuesta o resultados regresan en el buffer **RecvNCB** y el procedimiento Stub In se encarga de desempaquetar los resultados.

Para desempaquetar los argumentos, el Stub In toma información de la Tabla de Procedimientos Distribuidos y de la Cola de Estados de los Procesos.

La figura 3.5 muestra el esquema de del funcionamiento del Stub In en la estación destino.

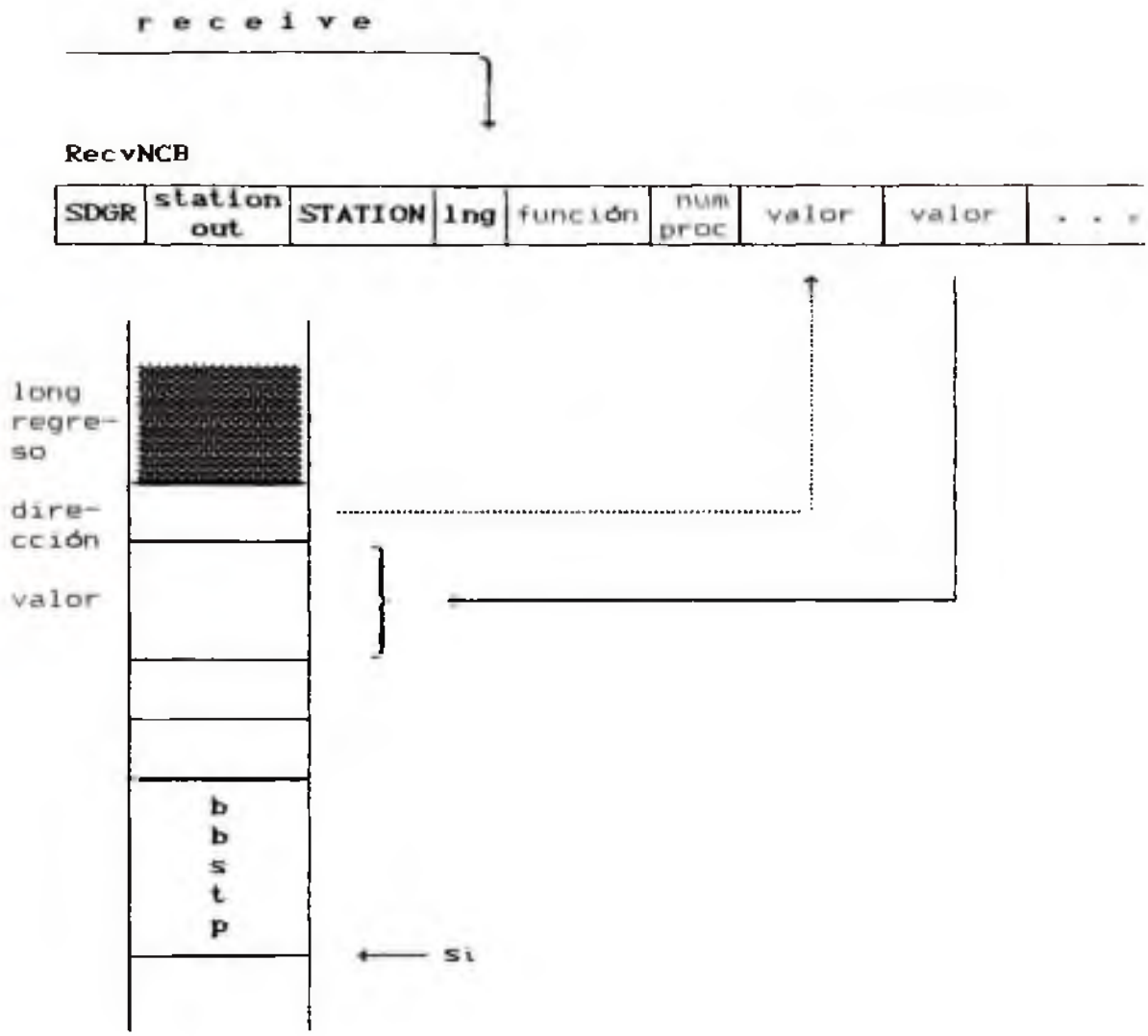


FIGURA 3.5 Esquema del funcionamiento del Stub In con función 1.

El Stub In se encarga de desempaquetar los argumentos y crear la instancia de llamado en el stack de variables del proceso solicitado. Obtiene información para el desempaquetamiento de la Tabla de Procedimientos Distribuidos y de la Cola de Estados de los Procesos.

Posteriormente crea una solicitud de procedimiento en la

Tabla de solicitudes.

La Tabla de solicitudes es una tabla con 10 campos que corresponden a los 10 procedimientos distribuidos (como máximo) que maneja el lenguaje. Estos campos son de tipo booleanos que indican lo siguiente:

solicitud [i] = True

Existe solicitud externa al procedimiento i

solicitud [i] = False

No existe solicitud externa al procedimiento i

El Stub In crea un solicitud en la Tabla de Solicitudes

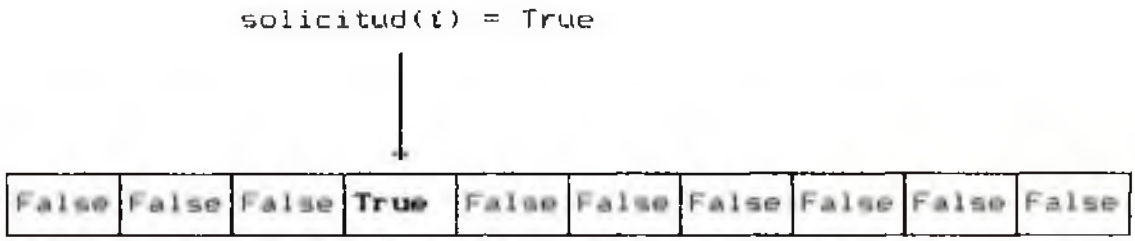


Tabla de solicitudes

Cuando el proceso externo que atiende a la solicitud externa del procedimiento i comparta el procesador, se ejecutará el procedimiento i. El pseudocódigo del proceso i es:

```
proc Proceso i:
   When solicitud(i) do
     Pi;
     stub_Out(2)
   end
end
```

El proceso i permanece bloqueado sí solicitud(i) = False.
El llamado a Stub Out con Función = 2, produce el regreso de resultados a la estación fuente.

IV. RED EDISON.

IV.1 Modelo de la Red.

En este capítulo se presenta el software de comunicación que permite al Sistema Edison trabajar en una pequeña red local por medio de un bus compartido y proveerá de servicios sobre dos niveles de red del modelo OSI (Open Systems Interconnection).
Estos niveles son: Nivel 1. Físico y Nivel 2. Enlace de datos

Nivel 2. Enlace de datos. Provee la transferencia de datos entre estaciones y detecta errores en la transferencia.
Nivel 1. Físico. Provee la transmisión transparente de cadenas de bits sobre el canal de comunicación.

Así, el software de comunicación lo constituyen los dos niveles más bajos de la OSI. El nivel físico define el medio de transmisión así como la velocidad y tipo de transmisión. Es transparente al usuario y es un servicio para el nivel de enlace de datos. El nivel de enlace de datos proporciona al usuario el envio de paquetes de datos entre dos estaciones sobre la red. Este servicio opera a traves de los siguientes comandos de datagramas:

INIT — La computadora que pide el servicio INIT, solicita de esta manera incorporarse a la red como una nueva estacion.

Send Datagram — Una estación puede enviar un datagrama a una estación destino mediante este servicio.

Send Broadcast Datagram. — Mediante este servicio se puede

enviar un datagrama a todas las estaciones.

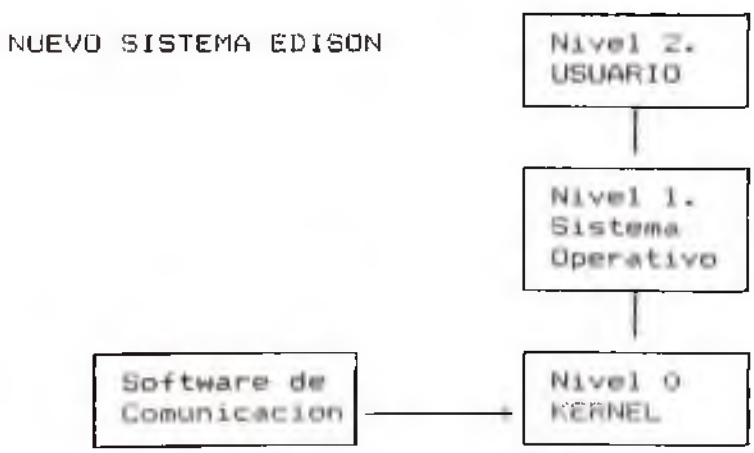
Receive Datagram. Una estación destino recibe un datagrama.

Receive Broadcast Datagram. Todas las estaciones reciben un datagrama enviado.

SHUT. Una estación puede salirse de la red mediante este servicio. Así, la red libera a la estación y la considera como una computadora independiente.

La manera que opera el nivel de enlace de datos para la realización de cada comando es mediante el envio de datagramas. Un datagrama es un paquete con información referente a: tipo de datagrama, estación destino (a quién se envia), estación fuente (quién lo envia), longitud del mensaje y por último el mensaje.

El nivel físico y nivel de enlace de datos forman el software de comunicación el cual se há desarrollado e emplementado en el nivel 0 del Sistema Edison. El software de comunicación se ha desarrollado en el lenguaje ensamblador Alva y forma parte del Kernel del Sistema. Anteriormente el Kernel ocupaba 6 páginas, ahora con la implementación del software de comunicación ocupa un total de 9 páginas.



El software de comunicación tiene como modelo una red de tipo bus usando el método de pooling, esto es: Una computadora es la encargada de controlar la red, determinando acceso del bus

a cada estación de la red. Con la asignación del bus a cada estación, esta la utiliza para enviar paquetes de máximo 256 caracteres. También se evita la colisión de paquetes obteniendose una transmisión eficiente pero también se consigue que la red sea lenta cuando el número de estaciones aumenta. La red se bloquea cuando la computadora que controla el bus también se bloquea.

El medio de comunicación o bus lo constituye un par de cables coaxiales que llegan a cada computadora mediante un conector RS232 serial. El tipo de comunicación de la red es asíncrona habilitandose cada rutina de comunicación por medio de interrupciones. La velocidad de transmisión ed de 9600 baud.

El esquema en que trabaja esta pequeña red es el siguiente: La computadora de control es la encargada de otorgar a cada estación el uso del medio de comunicación.
Cuando se le otorga el bus a una estación. esta utiliza enviando paquetes cuya longitud máxima es 256 caracters. El paquete recorre el bus y es captado por todas las estaciones, pero solamente es recibido por la estación destino y por la computadora de control que monitorea (despliega) el paquete enviado. Posteriormente la computadora de control asigna el bus a otra estación.
El tiempo que utilice el bus una estación dependerá del tamaño del paquete que desee transmitir. Si la información a transmitir es mayor de 256 caracteres, indica que estará dividida en paquetes de 256 caracteres y serán enviados cuando la computadora de control le asigne el bus.

La asignación del bus a todas las estaciones es de manera cíclica (una estación se le asigna nuevamente el bus después de asignarle al resto), también no existe prioridad de uso del bus.

El control y asignación del bus lo realiza el programa llamado PS (Pooling Station). Este programa corre en la computadora de control bajo el sistema MS-DOS versión 3.0 en adelante. Este programa es el único que no corre en el Sistema Edison ya que no fué necesario implementarlo porque al correr este programa la computadora no puede realizar otros procesos. Este programa está escrito en el lenguaje C y se describirá en IV.2.

A diferencia de la computadora de control, las estaciones corren bajo el Sistema Edison haciendo uso del software de comunicación para los comandos antes mencionados.
Tengase presente que el diseño de esta pequeña red no está destinada para la elaboración de servidores (archivos, procesos, impresión, etc.). El objetivo que se persigue en la red es poder transferir información relacionada con la ejecución de procesos o llamados a procesos remotos (RPC Remote Procedure Call).

IV.2 IMPLEMENTACION.

Todo paquete que transita la red tiene un protocolo general. Mediante este protocolo se distingue si el paquete contiene un mensaje a enviar, un reconocimiento o sincronización entre la computadora de control y una estación.
En terminos generales, cada paquete contiene los siguientes elementos:

inicio de protocolo	estación destino	datos	tipo de paquete	suma prueba

Inicio de protocolo. Es una bandera o marca que indica el inicio del paquete. El símbolo es FLAG.

Estación destino o remota. Es el número de la estación a la cual va dirigido el paquete. Esta medida es necesaria ya que el modelo de la red es de tipo bus.

Tipo de paquete. Describe el tipo de información que contiene el paquete. Existen cinco tipos de paquetes:

INF Indica que el paquete es de tipo información y es enviado por una estación fuente a la estación destino.
Indica además que posterior al código INF, el protocolo siguiente es: número de la estación fuente (quien envía el paquete), longitud de caracteres del mensaje que se envía, el mensaje que se envía.

ACK Este tipo de paquete es producido por la estación destino como manera de reconocimiento del paquete de tipo INF, que enviado por la estación fuente fué correctamente recibido.
El protocolo es acompañado del número de la estación fuente (estación que envía el reconocimiento).

REJ Este tipo de paquete indica que se recibió incorrecto el paquete. También lo envía la estación destino a la fuente (quien envió el paquete tipo INF).

ENQ Este tipo de paquete es enviado por la computadora de control a una estación especifica (estación remota), indicando que puede usar el medio de comunicación.

RSP Este tipo de paquete es enviado por la estación

actual que dispone del medio de comunicación pero no tiene paquetes para transmitir. Es usado como medio de respuesta a la computadora de control indicando que no necesita hacer uso del bus en ese momento.

Suma Prueba. Como elemnto final del paquete, se encuentra la suma prueba del paquete. Tiene como valor el complemento de la suma de todos los caracteres posteriores al caracter de inicio de protocolo.

La suma prueba es usado en la estación remota, como medida de veracidad del paquete recibido y así enviar a la estación fuente, un paquete de tipo ACK o REJ, indicando que el paquete se recibió correcto o incorrectamente.

En la descripción de la implementación, primeramente se describirá el controlador de la red y posteriormente el software de comunicación.

PROGRAMA POOLING STATION.

El controlador de la red es un pequeño programa encargado de asignar tiempo de servicio del bus a cada estación, además de desplegar en la pantalla la transferencia y reconocimiento de paquetes en cada estación. Este programa es llamado PS (Pooling Station).

Para asignar tiempo de uso del medio de comunicación a una estación, el programa PS envia a la estación el paquete de tipo ENQ. Es un paquete pequeño con la siguiente descripción.

FLAG	address	ENQ	address+ENQ
inicio de protocolo	estación destino	tipo de paquete	suma prueba

Cada estación recibe el paquete siendo solamente aceptado por la que coincide con address. Esta estación puede hacer uso del medio de comunicación: enviando paquetes a una determinada estación, reconociendo un paquete o enviando a la computadora de control el paquete tipo RSP. Cualquier tipo de paquete enviado por la estación es monitoreado por el programa PS el cual desplegará en pantalla cada acción realizada.

El programa PS inicia definiendo el tiempo de servicio asignado a todas la estaciones. Posteriormente llama a la rutina ComInit encargada de inicializar la comunicación asincrona de la interface serial RS-232, habilitando la interrupción 12.

Disponible la comunicación, comienza la disposición del bus a cada estación. Esta disposición se realiza mediante un ciclo limitado en el cual cada paso corresponde al servicio proporcionado a una estación.

La manera que el programa PS asigna el bus a una estación es mediante el envio de un paquete de tipo ENQ con su dirección o número de estación, luego esperará respuesta de la estación a fin de desplegar el servicio utilizado. En caso de no existir respuesta, la estación estará inactiva y se le desplegará en la pantalla como estación en TMOUT.

Como se ha presentado, el programa PS es pequeño. Más que un controlador es un asignador de tiempo que monitorea cada paquete en la red. El programa termina cuando el usuario oprime la tecla de ESCAPE, entonces el programa deja de asignar tiempos y la red deja de existir.

SOFTWARE DE COMUNICACION.

El software de comunicación lo constituye un conjunto de rutinas cuya finalidad es sincronizar la comunicación con el programa PS ubicado en la computadora de control. Tanto el software de comunicación como el programa PS forman el modelo de red tipo bus usando el método de pooling. El software de

comunicación es implementado en cada estación y lo componen las siguientes rutinas:

getvect — La rutina getvect trae el contenido o dirección de rutina del vector de interrupciones. Sus parametros son getvect (no. interrupción, asignación). Estos parametros se pasan a traves del stack.

setvect — Deposita la dirección de la rutina que atenderá a la interrupción enmarcada. Sus parametros son setvect (no. interrupción, asignación).

Newtimer — Rutina de tipo interrupción (se ejecuta cada pulso de reloj).
Esta rutina realiza un decremento en cada pulso de reloj de los temporizadores de transmisión y recepción (si se encuentran activos).

Com Init — Rutina que inicializa el tipo de transmisión. Habilita la interrupción 12 (RS-232) y una señal de transmisión de 8 bit y un stop bit.

ComRest — Tiene como objetivo restaurar la direcciones de la interrupción 12.

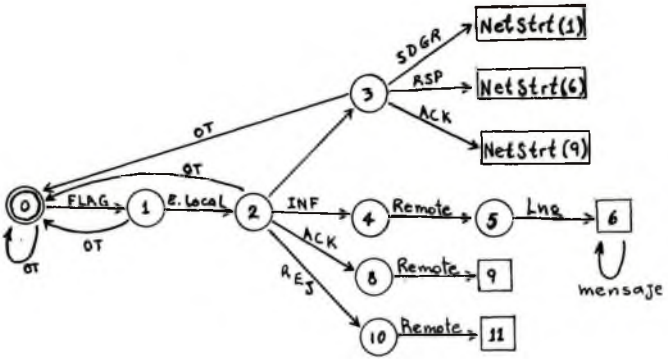
NetStrt — Inicia la estación para transmisión de datos. Envia el inicio del protocolo (símbolo FLAG).

NewRS232 — Rutina de tipo interrupción. Se activa cuando existe una interrupción número 12, la cual se presenta cuando existe transmisión o recepción de datos.

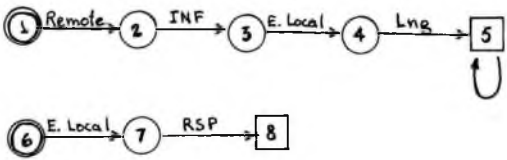
El procedimiento que realiza el software de comunicación en la recepción de paquetes está determinada por dos automatas de estados. El primer autómata es llamado autómata de entrada que se ocupa de la recepción de datos, el segundo autómata es

de salida y se ocupa de la transmisión de datos.
El autómata de entrada habilita al autómata de salida cuando llama a la rutina NetStrt con un parametro que indicará en qué estado inicial comenzará el automata de salida.
El autómata de entrada se habilita por cada interrupción o dato que se presenta en el puerto de datos. El dato se analiza y se determina si el autómata pasa al siguiente estado o se regresa al estado inicial.
Como se aprecia en la gráfica (siguiente hoja), solo existe un estado inicial (estado 0) y no se puede seguir al siguiente estado a menos que el dato recibido sea el inicio del protocolo, de lo contrario estará siempre en el estado inicial.
El autómata de salida es habilitado por el autómata de entrada cuando este se encuentra en el estado 3. Como su nombre lo indica, el autómata de salida se utiliza para la transmisión del protocolo: transmitir un mensaje a una estación, enviar un reconocimiento a una estación o contestar a la computadora de control la signación del bus.
El autómata de salida tiene tres estadbs iniciales que indican cada tipo de transmisión. El estado inicial es seleccionado por el autómata de entrada cuando llama a la rutina NetStrt con el estado incial.
La rutina NetStrt se encarga de enviar el código del inicio de protocolo (FLAG = 126), y asignar la variable sstate = estado inicial. Esta variable indica el estado actual del autómata de salida.

La siguiente figura muestra el flujo de estados de cada autómata.



AUTOMATA DE ENTRADA



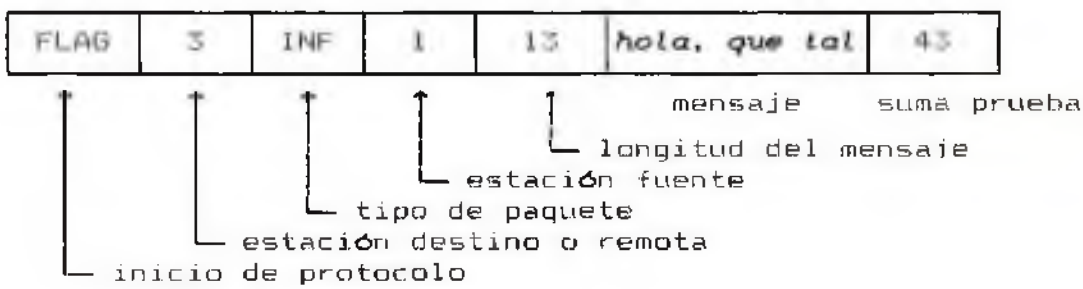
AUTOMATA DE SALIDA

Estado Inicial Estado Final OT Otro caso

Para aclarar el funcionamiento de ambos autómatas se describirá un ejemplo.

Ejemplo:

Supongamos que la estación 1 desea enviar el siguiente mensaje "*hola, que tal*" a la estación 3.
Cualquier información que se transfiere (recepción, transmisión), es representado por un paquete. Así, el paquete con el mensaje "*hola, que tal*", que la estación 1 envia es el siguiente:



Este paquete viajará por el medio de transmisión (cuando la estación 1 tenga el tiempo de servicio) y llegará a todas las estaciones. En cada estación se habilitará el autómata de entrada.
Sí los caracteres del paquete no se han perdido o transformado al viajar por el medio de transmisión, se tendrá en la estación 3 el recorrido del autómata con los siguientes estados.

0 1 2 3 4 5 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6

Para las demás estaciones el recorrido del autómata es:

0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

También el programa PS (en la computadora de control) recibirá paquete y desplegará en pantalla la siguiente información.

1 3 INF 1 13 hola, que tal

Cuando la estación 3 tenga el tiempo de servicio del medio, enviará el paquete de tipo ACK a la estación 1.

FLAG	1	ACK	3	1 + ACK + 3

En caso que el paquete no se haya recibido correctamente, la estación 3 enviará el paquete de tipo REJ a la estación 1.

FLAG	1	REJ	3	1 + REJ + 3

Para ambos casos, el programa PS desplegará en la pantalla (computadora de control), la información:

3 1 ACK

La estación 3 envia a la estación 1 un reconocimiento.

3 1 REJ

La estación 3 envia a la estación 1 la señal de paquete no reconocido.

Cuando la estación 1 envia el mensaje "*hola, que tal*" a la estación 3, se considerá la creación del paquete el cual ya se ha descrito anteriormente:

FLAG	3	INF	1	13	*hola, que tal*	43

Este paquete es enviado por el autómata de salida de la siguiente manera:

Se comienza cuando la computadora de control asigna el tiempo de servicio a la estación 1. Las señales llegan a la estación 1 y recorren el autómata de entrada hasta el estado 3. Como la estación 1 tiene un mensaje de envio a la estación 3, se hace un llamado a la rutina NetStrt (estado 3) con el parametro inicio de estado igual a 1.
La rutina NetStrt envia el código de inicio de protocolo (FLAG = 126), posteriormente el autómata de salida se habilita en el estado inicial igual a 1, enviando el resto del paquete con los siguientes estados:

2 3 4 5 5 5 5 5 5 5 5 5 5 5 5 5

MILAN Es la rutina principal en el software de comunicación encargada de aceptar los comandos de servicios por el usuario. Los comandos son los siguientes:

INIT Inicializa la estación bajo un número asignado por el usuario.

SDGR Send Datagram. Transmite un datagrama o paquete a una estación remota.

RDGR Receive Datagram. Recibe un datagrama enviado por una estación.

SENDBRD Send BroadCast Datagram. Transmite un datagrama a todas las estaciones.

RECBRD Receive BroadCast Datagram. La estación local recibe un datagrama sin validar la estación destino.

SHUT Finalización. La red libera a la estación y y la considera como computadora independiente.

MILAN recibe la petición del comando bajo un

parametro ***paquete*** de tipo NCB. La solicitud del servicio es invocado como: MILAN (***paquete***)
El tipo NCB es una estructura que en Edison tiene la siguiente forma.

```
record  NCB  ( Command: int;
               RemAddr: int;
               LocAddr: int;
               Lng: int;
               TmOut_: int;
               Complete: int;
               Buffer: line
             )
```

Command es el tipo de servicio que se requiere.
RemAddr es la estación remota a la cual va dirigido el servicio.
LocAddr es la estación local o fuente quien envía el servicio.
Lng es la longitud o numero de caracteres que corresponde al mensaje.
TmOut_ es el temporizador anexado al servicio.
Complete es la respuesta enviada por MILAN como respuesta al servicio. Estas pueden ser OK, ERROR, WAITING.
Buffer es un vector de tipo linea donde se deposita el mensaje.

En el ejemplo anterior. Cuando la estación 1 envia el mensaje "***hola, que tal***" a la estación 3, se requiere que la estacion 1 llene la estructura ***paquete*** de la siguiente forma:

```
paquete.Command := SDGR;
paquete.RemAddr := 3;
paquete.LocAddr := 1;
paquete.Lng := 13
paquete.TmOut_ := 15;
```

```
paquete.Complete := WAITING;
paquete.Buffer := line('hola, que tal');
```

Luego se llama a la rutina MILAN con la estructura.

```
MILAN (paquete);
```

La cual enviará el paquete a la estación 3.

La rutina MILAN es la unica que el usuario puede utilizar del conjunto de rutinas del software de comunicación. Como se ha mencionado anteriormente el proposito del software de comunicación es como medida de servicio para la ejecución de procesos remotos, de ninguna manera se pretende utilizarlo para soportar servidores.

V. R E S U L T A D O S

Se presenta un programa distribuido productor-consumidor compuesto de procesos α. En el programa existen 3 procesos α, "producer", "buffer" y "consumer".

El proceso "producer", definido en la estación 1 se encarga de leer nombres de la terminal y enviarlo al proceso "buffer" definido en la estación 2. Este proceso se encarga de almacenar los nombres en un arreglo. Otro proceso llamado "consumer" definido en la estación 3, se encarga de pedir un nombre del arreglo y escribir el nombre en la terminal.

El código fuente de los tres procesos se presentan en las sguientes páginas. Posteriormente se presenta el código Edison de los tres procesos. Este código lo interpreta el Kernel en cada estación.

El código Edison que genera el compilador Edison_D son primitivas generales ya que no involucra ningún tipo de red. sin embargo, el Kernel al interpretar las primitivas de tipo distribuido, realiza llamados al Stub Out y Stub In, que utilizan la red MILAN.

Como puede notarse, solo se necesitan pequeños cambios al Stub Out y Stub In para que el sistema Edison se integre en otra red.

PROCESO PRODUCER

```
const namelength = 12 "characters"
array name [1:namelength] (char)

proc prefix(
  proc display(value: char);
  proc accept(var value: char);
  proc readname(proc read(var c: char); var value: name);
  proc writename(proc write(c: char); value: name)
          )

  station  1
  remote proc deposita (mensaje: name) on 2
  remote proc enviar (var mensaje: name) on 2
  var nombre: name; band: bool

  begin            "producer"
    band := true;
    while band do
      readname(accept, nombre);
      deposita(nombre)
    end
  end
```

PROCESO BUFFER

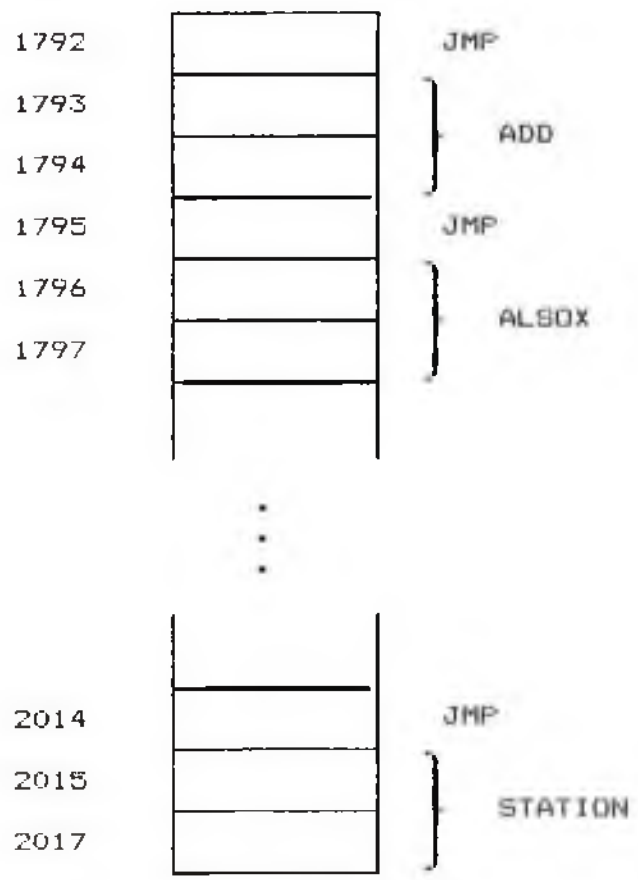
```
const namelength = 12 "characters"
array name [1:namelength] (char)
proc prefix(
  proc display(value: char);
  proc accept(var value: char);
  proc readname(proc read(var c: char); var value: name);
  proc writename(proc write(c: char); value: name)
          )
  station 2
  entry proc deposita (mensaje: name)
  entry proc enviar (var mensaje: name)

  module "buffer"
    const n = 20
    array vector[1:n] (name)
    var   buffer: vector;
          head, tail, size: int
    * proc deposita (x: name)
      begin
        when size < n do
          buffer[tail] := x;
          size := size +1;
          tail := (tail + 1) mod n
        end
      end
    * proc envia (var x: name)
      begin
        when size > 0 do
          x := buffer[head];
          size := size - 1;
          head := (head + 1) mod n
        end
      end
    begin head := 0; tail := 0; size := 0 end
  begin skip end
```

PROCESO CONSUMER

```
const namelength = 12 "characters"
array name [1:namelength] (char)

proc prefix(
  proc display(value: char);
  proc accept(var value: char);
  proc readname(proc read(var c: char); var value: name);
  proc writename(proc write(c: char); value: name)
          )

  station 3
  remote proc deposita (mensaje: name) on 2
  remote proc enviar (var mensaje: name) on 2

  var nombre: name; band: bool

  begin
    band := true;
    while band do
      envia (nombre);
      writename (display, nombre)
    end
  end
```

El código generado para los tres procesos se describen en las páginas siguientes. Esta descripción se logra por medio de un programa llamado **desp4**, que toma el código generado (código Edison), por el compilador y describe las primitivas correspondientes.
El programa empieza mediante una dirección relativa (a partir de 0). Cada instrucción o datos es representado en 2 bytes de memoria (1 word). Toda instrucción tiene un valor entre 0 y 74, más un desplazamiento de 1792, que corresponde a la dirección física donde se interpreta cada intrucción.

A partir de la dirección 1792 existen saltos a cada rutina que atiende a la primitiva.



```
ARCHIVO: producer
Codigo Generado
Compilador Edison-PC Distribuido
Paso 4

    Dir                Codigo
  Relativa  ----- Generado -----

  [  0]          152
  [  2]         1792 + 3 * 71        @initial4
  [  4]         1792 + 3 * 39        @procedure4
  [  6]          162
  [  8]           26
  [ 10]          174
  [ 12]           58
  [ 14]         1792 + 3 * 74        @station4
  [ 16]            1
  [ 18]         1792 + 3 * 72        @remote4
  [ 20]            1
  [ 22]            0
  [ 24]            1
  [ 26]            1
  [ 28]           24
  [ 30]         1792 + 3 * 72        @remote4
  [ 32]            2
  [ 34]            0
  [ 36]            1
  [ 38]            2
  [ 40]           24
  [ 42]         1792 + 3 * 70        @execute4
  [ 44]         1792 + 3 * 58        @localvar4
  [ 46]           34
  [ 48]         1792 + 3 *  6        @constant4
  [ 50]            1
  [ 52]         1792 + 3 * 53        @elemassign4
  [ 54]         1792 + 3 * 57        @localvalue4
  [ 56]           34
  [ 58]         1792 + 3 * 10        @do4
  [ 60]           88
  [ 62]         1792 + 3 * 22        @instance4
  [ 64]            0
  [ 66]         1792 + 3 * 35        @paramarg4
  [ 68]         -120
  [ 70]         1792 + 3 * 67        @strconst4
  [ 72]           10
  [ 74]          110
  [ 76]          111
  [ 78]          109
  [ 80]           98
  [ 82]          114
```

```
[ 84]          101
[ 86]           32
[ 88]           61
[ 90]           32
[ 92]           46
[ 94]         1792 + 3 *  4        @blank4
[ 96]           70
[ 98]         1792 + 3 * 22        @instance4
[100]            0
[102]         1792 + 3 * 36        @paramcall4
[104]          -20
[106]         1792 + 3 * 22        @instance4
[108]            0
[110]         1792 + 3 * 35        @paramarg4
[112]         -112
[114]         1792 + 3 * 58        @localvar4
[116]           10
[118]         1792 + 3 * 22        @instance4
[120]            0
[122]         1792 + 3 * 36        @paramcall4
[124]          -36
[126]         1792 + 3 * 58        @localvar4
[128]           10
[130]         1792 + 3 * 44        @value4
[132]           24
[134]         1792 + 3 * 73        @remotecall4
[136]            1
[138]           24
[140]            2
[142]         1792 + 3 * 11        @else4
[144]          -88
[146]         1792 + 3 * 14        @endproc4
[148]         1792 + 3 * 12        @encode4
[150]           74
```

```
ARCHIVO: buffer
Codigo Generado
Compilador Edison-PC Distribuido
Paso 4

    Dir                 Codigo
  Relativa  ----- Generado -----

  [  0]          320
  [  2]         1792 + 3 * 71        @initial4
  [  4]         1792 + 3 * 39        @procedure4
  [  6]          162
  [  8]          486
  [ 10]            4
  [ 12]           58
  [ 14]         1792 + 3 * 74        @station4
  [ 16]            2
  [ 18]         1792 + 3 * 69        @entry4
  [ 20]           32
  [ 22]            1
  [ 24]            0
  [ 26]            1
  [ 28]            1
  [ 30]           24
  [ 32]         1792 + 3 * 69        @entry4
  [ 34]          136
  [ 36]            2
  [ 38]            0
  [ 40]            1
  [ 42]            2
  [ 44]           24
  [ 46]         1792 + 3 * 18        @goto4
  [ 48]          118
  [ 50]         1792 + 3 * 68        @defentry4
  [ 52]           24
  [ 54]            0
  [ 56]           26
  [ 58]            1
  [ 60]           71
  [ 62]         1792 + 3 * 47        @when4
  [ 64]         1792 + 3 * 63        @outervalue4
  [ 66]          494
  [ 68]         1792 + 3 *  6        @constant4
  [ 70]           20
  [ 72]         1792 + 3 * 24        @less4
  [ 74]         1792 + 3 * 10        @do4
  [ 76]           82
  [ 78]         1792 + 3 * 64        @outervar4
  [ 80]           10
  [ 82]         1792 + 3 * 63        @outervalue4
```

```
[ 84]          492
[ 86]         1792 + 3 * 21        @index4
[ 88]            1
[ 90]           20
[ 92]           24
[ 94]           73
[ 96]         1792 + 3 * 58        @localvar4
[ 98]          -24
[100]         1792 + 3 * 44        @value4
[102]           24
[104]         1792 + 3 *  3        @assign4
[106]           24
[108]         1792 + 3 * 64        @outervar4
[110]          494
[112]         1792 + 3 * 63        @outervalue4
[114]          494
[116]         1792 + 3 *  6        @constant4
[118]            1
[120]         1792 + 3 *  0        @add4
[122]           74
[124]         1792 + 3 * 53        @elemassign4
[126]         1792 + 3 * 64        @outervar4
[128]          492
[130]         1792 + 3 * 63        @outervalue4
[132]          492
[134]         1792 + 3 *  6        @constant4
[136]            1
[138]         1792 + 3 *  0        @add4
[140]           75
[142]         1792 + 3 *  6        @constant4
[144]           20
[146]         1792 + 3 * 27        @modulo4
[148]           76
[150]         1792 + 3 * 53        @elemassign4
[152]         1792 + 3 * 11        @else4
[154]            8
[156]         1792 + 3 * 46        @wait4
[158]          -94
[160]         1792 + 3 * 15        @endwhen
[162]         1792 + 3 * 14        @endproc4
[164]         1792 + 3 * 18        @goto4
[166]          118
[168]         1792 + 3 * 68        @defentry4
[170]            2
[172]            0
[174]           26
[176]            2
[178]           80
[180]         1792 + 3 * 47        @when4
[182]         1792 + 3 * 63        @outervalue4
[184]          494
```

```
[186]        1792 + 3 *  6        @constant4
[188]           0
[190]        1792 + 3 * 19        @greater4
[192]        1792 + 3 * 10        @do4
[194]          82
[196]        1792 + 3 * 57        @localvalue4
[198]          -2
[200]        1792 + 3 * 64        @outervar4
[202]          10
[204]        1792 + 3 * 63        @outervalue4
[206]         490
[208]        1792 + 3 * 21        @index4
[210]           1
[212]          20
[214]          24
[216]          82
[218]        1792 + 3 * 44        @value4
[220]          24
[222]        1792 + 3 *  3        @assign4
[224]          24
[226]        1792 + 3 * 64        @outervar4
[228]         494
[230]        1792 + 3 * 63        @outervalue4
[232]         494
[234]        1792 + 3 *  6        @constant4
[236]           1
[238]        1792 + 3 * 41        @subtract4
[240]          83
[242]        1792 + 3 * 53        @elemassign4
[244]        1792 + 3 * 64        @outervar4
[246]         490
[248]        1792 + 3 * 63        @outervalue4
[250]         490
[252]        1792 + 3 *  6        @constant4
[254]           1
[256]        1792 + 3 *  0        @add4
[258]          84
[260]        1792 + 3 *  6        @constant4
[262]          20
[264]        1792 + 3 * 27        @modulo4
[266]          85
[268]        1792 + 3 * 53        @elemassign4
[270]        1792 + 3 * 11        @else4
[272]           8
[274]        1792 + 3 * 46        @wait4
[276]         -94
[278]        1792 + 3 * 15        @endwhen
[280]        1792 + 3 * 14        @endproc4
[282]        1792 + 3 * 58        @localvar4
[284]         490
[286]        1792 + 3 *  6        @constant4
```

```
[288]          0
[290]       1792 + 3 * 53        @elemassign4
[292]       1792 + 3 * 58        @localvar4
[294]        492
[296]       1792 + 3 *  6        @constant4
[298]          0
[300]       1792 + 3 * 53        @elemassign4
[302]       1792 + 3 * 58        @localvar4
[304]        494
[306]       1792 + 3 *  6        @constant4
[308]          0
[310]       1792 + 3 * 53        @elemassign4
[312]       1792 + 3 * 70        @execute4
[314]       1792 + 3 * 14        @endproc4
[316]       1792 + 3 * 12        @encode4
[318]         92
```

ARCHIVO: consumer
Codigo Generado
Compilador Edison-PC Distribuido
Paso 4

Dir Relativa	Codigo Generado	
[0]	108	
[2]	1792 + 3 * 71	@initial4
[4]	1792 + 3 * 39	@procedure4
[6]	162	
[8]	26	
[10]	40	
[12]	58	
[14]	1792 + 3 * 74	@station4
[16]	3	
[18]	1792 + 3 * 72	@remote4
[20]	1	
[22]	0	
[24]	1	
[26]	1	
[28]	24	
[30]	1792 + 3 * 72	@remote4
[32]	2	
[34]	0	
[36]	1	
[38]	2	
[40]	24	
[42]	1792 + 3 * 70	@execute4
[44]	1792 + 3 * 5B	@localvar4
[46]	34	
[48]	1792 + 3 * 6	@constant4
[50]	1	
[52]	1792 + 3 * 53	@elemassign4
[54]	1792 + 3 * 57	@localvalue4
[56]	34	
[58]	1792 + 3 * 10	@do4
[60]	44	
[62]	1792 + 3 * 58	@localvar4
[64]	10	
[66]	1792 + 3 * 73	@remotecall4
[68]	2	
[70]	2	
[72]	2	
[74]	1792 + 3 * 22	@instance4
[76]	0	
[78]	1792 + 3 * 35	@paramarg4
[80]	-120	
[82]	1792 + 3 * 5B	@localvar4

```
[ 84]          10
[ 86]        1792 + 3 * 44        @value4
[ 88]          24
[ 90]        1792 + 3 * 22        @instance4
[ 92]          0
[ 94]        1792 + 3 * 36        @paramcall4
[ 96]         -24
[ 98]        1792 + 3 * 11        @else4
[100]         -44
[102]        1792 + 3 * 14        @endproc4
[104]        1792 + 3 * 12        @encode4
[106]          73
```

C O N C L U S I O N E S

Primeramente se partió iniciando el estudio del Sistema Edison correspondiente al Sistema Operativo, Kernel y el compilador Edison.

De acuerdo al lenguaje Edison se tomó la alternativa por parte del asesor Dr. Jan Janecek, de estudiar la posibilidad de implementar un lenguaje distribuido en el lenguaje Edison. Esta alternativa se hizo posible debido que los conceptos de los procesos distribuidos de Hansen se adaptaron al lenguaje mediante un diseño enfocado a conservar la semántica de Edison.

Posteriormente al diseño, se realizaron las modificaciones necesarias al compilador Edison, transformandolo en Edison_D. Sin embargo, también era necesario describir el diseño de RPC el cual sería implantado en el Kernel. En esta parte fué necesario estudiar las zonas de memoria que el Kernel describe para los procesos.

En lo que concierne al software de comunicación se tomó como base la red MILAN [Jan Janecek] y se instaló en el kernel del sistema bajo el lenguaje Alva. Para tal proposito se creó un nuevo módulo del kernel llamado Kernel_ext que permite que el código del kernel supere los 3072 words que determina el sistema Edison.

B I B L I O G R A F I A

1. Andrews, G.R., Schneider, F. B. "Concepts and notations for concurrent programming", ACM Comp. Surveys 15,1 (Mar. 1983) pp. 3-43.
2. Birrell, A., Nelson, B., "Implementing remote procedure calls", ACM TOCS, 2,1 (Feb. 1984), 39-59.
3. Brinch Hansen, P. "Distributed processes: A concurrent programming concept." Comm. ACM 21,11 (Nov. 1978), 934-941.
4. Brinch Hansen, P. "Edison: A multiprocessor language." Softw. Pract. Exper. 11,4 (Apr. 1981), 325-361.
5. Brinch Hansen, P. "Programming a Personal Computer", Prentice Hall, Englewood Cliffs, New Jersey. 1983.
6. Champine A. George, "Distributed computer systems", North Holland Publishing Company, 1980.
7. Dijkstra, E. W. "Guarded commands, nondeterminacy, and formal derivation of programs", Comm. ACM 18,8 (Aug. 1975) 453-57.
8. Dubnicki, C., Madey, J. and Wygladala, W. "Edison-N: an Edison implementation for a network fo microcomputers" Softw. Pract. Exper. 18,4 (Apr. 1988), 349-363.
9. Hoare, C. A. R. "Monitors: An Operating system structuring concept", Comm. ACM 17, 10 (Oct. 1974), 549-557.
10. Liu Yu-cheng, Gibson Glenn A., "Microcomputer systems: The 8086/8088 family", Prentice Hall, Englewood Cliffs, New Jersey, 1986.
11. Morse, Stephen P. "The 8086/8088 Primer", Hayden Book C., Inc. Rochelle Park, New Jersey, 1982.

El jurado designado por la Sección de Computación del Departamento de Ingeniería Eléctrica del Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, Aprobó esta tesis el 21 de junio de 1990.

Dr. Jan Janecek Hyan

Dr. Mario Albarrán Figueroa

M. en C. César Guzmán Rentería