ISSN 1870-4069

Actividades colaborativas en sistemas distribuidos construidos con Erlang

Gerardo Carreño-García[1], Manuel Hernández-Gutiérrez[2]

[1] Banco de México, Ciudad de México, México
[2] Universidad Tecnológica de la Mixteca, Huajuapan de León, Oaxaca, México
gcarreno@banxico.org.mx, manuelhg@mixteco.utm.mx

Resumen. Este escrito presenta una formulación gradual de un sistema distribuido para la realización de labores que utilizan varios posibles nodos computacionales. El sistema es analizado a nivel del paradigma de concurrencia de *paso de mensajes* e implementado en el lenguaje de programación funcional y concurrente Erlang. El diseño del sistema resultante se basa en una metodología que comienza con la identificación de una partición de la labor original, continúa con la posible asignación de sublabores a nodos computacionales (aprovechando posibles oportunidades de colaboración inteligente) y, finalmente, por una estructuración de una solución final basada en subsoluciones.

Palabras clave: cómputo distribuido, colaboración, Erlang, paso de mensajes.

Collaborative Activities in Distributed Systems Built in Erlang

Abstract. This paper presents a gradual formulation of a distributed system for the accomplishment of tasks that use several possible computational nodes. The system is analyzed at the paradigm level of concurrent message passing and implemented in Erlang, a functional and concurrent language programming. The design of the resulting system is based on a methodology that begins with the identification of a partition of the original work, continues with the possible assignment from subtasks to computational nodes (taking advantage of possible opportunities for intelligent collaboration) and, finally, by structuring a final solution based on sub-solutions.

Keywords: distributed computing, collaboration, Erlang, message passing.

1. Introducción

Los sistemas distribuidos representan una gran oportunidad para realizar labores complicadas en paralelo o concurrentemente, siendo así que se abre la

puerta al tratamiento de labores que pueden desarrollarse en diversos nodos computacionales tanto por razones de eficiencia como de formulación inherentemente distribuida (como sería el caso de un conjunto de robots que realizan labores de salvamento en regiones geográficamente dispersas). En este artículo presentamos un sistema distribuido que realiza ejecuciones distribuidas de varios programas que incorporan algunas técnicas de subdivisión de una labor en sublabores. Estos programas, que pueden considerarse *agentes*, son implementados en el lenguaje de programación funcional y concurrente Erlang [1,2,7].

Enmarcaremos nuestras formulaciones téoricas en el paradigma de paso de mensajes para cómputo distribuido (particularmente, modelado matemáticamente como un *cálculo de actores*; ver [19]). Para algunas codificaciones se ha utilizado una metodología transformacional de programas funcionales [6,17], aprovechando el aspecto funcional de los programas de Erlang. Para el caso de programación distribuida entre agentes se sigue una filosofía del establecimiento de *protocolos*, *mensajes* y *diálogos* ([2]). Los protocolos indican qué se puede válidamente comunicar entre agentes, los mensajes seguirán estos protocolos y contendrán información relevante, y los diálogos serán sucesiones de mensajes entre diversos emisores y receptores.

Aplicaremos nuestros ejemplos generales a la especificación (descripción) e implementación de un sistema distribuido basado en agentes que, en una parte, gestionan tareas asignadas, y en otra, colaboran entre sí para llevar a cabo algunos objetivos (apoyándonos en algunos puntos presentados en [8] y en [4]. Se considera que una aportación medular de este trabajo es pasar de los diseños teóricos a implementaciones concretas vías programas de Erlang, así logrando que Erlang sea un de laboratorio computacional que hace de puente entre las conceptualizaciones teóricas y las técnicas de programación concurrente. Siendo Erlang por naturaleza *asíncrono*, enfatizaremos qué hacer cuando por alguna razón se requiene sincronía. Debido a las primitivas de concurrencia ya existentes *de facto* en Erlang, manejamos la hipótesis de que éste lenguaje de propósito general es preferible a otros.

Se supondrá cierto conocimiento básico de programación funcional, de teoría de grafos, y las características generales de los problemas de concurrencia, aunque en la parte distribuida con Erlang se entrará prácticamente de lleno (para lo cual ayudaría ver los artículos divulgativos [12,13]). A continuación, aplicaremos nuestros desarrollos parciales a la puesta en marcha de un sistema distribuido que trata con agentes inteligentes como procesos hospedándose en *nodos-e*, en donde la parte de *inteligencia* será colocada en la manera de distribuir las labores asignadas, promoviendo un estilo colaborativo de solución entre los agentes participantes.

2. Erlang: concurrencia y programación funcional

En esta sección presentamos algunas ideas que hacen del lenguaje de programación Erlang una adecuada herramienta de pensamiento aplicado a los sistemas distribuidos. Veremos que el punto de apoyo principal teórico es el de *sistemas*

distribudos con procesos asíncronos y paso de mensajes, la creación de procesos ligeros, y la posibilidad tersa de escalabilidad.

El punto de entrada. Erlang es un lenguaje de programación de propósito general con algunas primitivas directamente relacionadas con la teoría de un modelo distribuido de cómputo basado en paso de mensajes. Estas primitivas pueden en principio realizar tres tareas básicas: generar *procesos* de tipo ligero (ajenos y en convivencia con los procesos del sistema operativo en uso), *enviar mensajes*, y *recibir mensajes*. Los procesos se hospedan en un *nodo-e*, que es un un intérprete del lenguaje Erlang ejecutándose en una computadora.

Erlang tiene un estilo de *programación funcional*, pero a la fecha no se ha logrado que encaje en algún modelo teórico existente funcional y distribuido (aunque un intento digno de citarse es [15], basándose en el *cálculo pi*). De todas formas, con una metodología disciplinada que comience con una adecuada especificación, un desarrollo de programación apegado a esta especificación, y una implementación que satisfaga también la especificación, el problema de conocer e identificar la semántica del programa final se aminora.

La especificación de un problema y su solución en Erlang. Modelar sistemas distribuidos que se apeguen a una especificación de la manera más fiel posible, es algo preferible antes que una directa implementación [9,19]. No obstante, una de las banderas enarboladas desde hace tiempo por la comunidad de programadores funcionales es que frecuentemente los programas funcionales *actúan* como especificaciones [5]. De requerirse que estas especificaciones funcionales se apeguen de manera más directa a una ejecución en cierto modelo computacional, se pueden utilizar herramientas del tipo de *transformación de programas* [16]. Estas transformaciones son preservadoras de las semánticas iniciales, y con ello puede lograrse adecuadas implementaciones en el sentido de eficiencia. Cuando tales técnicas no puedan aplicarse, las herramientas de construcción de modelos (*model checking*, [10,11]) pueden ser oportunamente atraídas.

El núcleo de conceptos de Erlang es invariante a diversa escala. Erlang tiene una propiedad para realizar investigación de sistemas distribuidos: Es *tersamente* escalable y puede ser utilizado como un puente entre teoría y práctica, haciendo el papel de *laboratorio de prueba* (*sandbox*) para percibir tempranamente problemas de implementación, y una vez resueltos, facilitar la construcción de programas solución, para hacer pasar estos programas prácticamente sin cambios al campo industrial. Para ello, se comienza con un sistema de procesos sobre un nodo; posteriormente, en sistemas operativos tales como Linux, es posible abrir varias consolas, en donde cada consola es tratada como un nodo (computadora) independiente. En un último paso, la escalabilidad queda asegurada ya que se podrían ahora, con pequeños cambios, utilizar el sistema sistema distribuido (en apariencia, de "juguete") hecho en una sola computadora sobre varias otras (eso sí, de preferencia con conexiones que se suponen estables). Por lo demás, este artículo sigue la investigación de relacionar agentes con Erlang [18].

3. Vectores para rastreo de cómputo distribuido

Consideramos ahora un problema que requiere para su solución una subdivisión de labores, en subtareas. Si preliminarmente trabajamos con un sistema centralizado, supongamos que requerimos sincronía, monitoreada por un agente en particular, encargado de monitorear qué subtareas se han terminado y cúales no. Para el monitoreo de la realización de subtareas, necesitaremos un concepto auxiliar de *vector de terminación*. Definimos un *vector de terminación de tamaño n* que está puesto en 0s al inicio de toda computación distribuida, con cada entrada i ($1 \leq i \leq n$) indicando el estado de terminación de un proceso (llamado en adelante *trabajador*) i-ésimo. Estos vectores pueden tener un índice de terminación, indicando históricamente quién ha terminado su labor y en qué momento. Además, pueden servir para llevar un rastreo también histórico de qué tan eficaz y eficiente es un trabajador dado. Tales actividades son por el momento asignadas a un proceso especial que llamaremos en adelante `agenteOrd` (transitando de lleno, en un futuro, hacia el concepto de *agente*).

Por ejemplo, en la secuencia de vectores de terminación:

$$\begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1_2 & 1_1 & 1_3 \end{bmatrix}$$

se hizo un registro de qué trabajadores terminaron y cuando. Para el caso de una vigilancia parcial, se tendría:

$$\begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1_2 & 1_1 & 0 \end{bmatrix}$$

indicando que todavía se espera la respuesta del trabajador número 3. Cabe notar que, por diseño, es posible que agentes descentralizados estén equipados con estas capacidades de monitoreo, si así fuera necesario.

La función que apoyaría al proceso A (`agenteOrd`) (en posible compañía de un *operador*, que puede ser un humano) a verificar la terminación de los asignamientos dados para el caso de dos trabajadores sería, para una posible implementación, así:

```
procesoA({Tarea,{X,Y}}) ->
{_,V}={_,{X,Y}},
if {X,Y}=={1,1} -> io:format(''Tarea terminada ~n'',[]),
      %notificación al operador
receive
                    {terminada,ProcesoB} -> X=1,
                              procesoA({Tarea,{X,Y}});
                    {terminada, ProcesoC} -> Y=1,
                     procesoA({Tarea,{X,Y}})
                     %otros mensajes a recibir...
end.
```

en donde se supone de cada proceso B y C tiene la capacidad para enviar el mensaje de terminación; es posible con la construcción de Erlang `after` también enviar mensajes al operador de la *no terminación* (hasta ese punto del tiempo)

de la tarea intermitente, para un intervalo dado. La sincronía programada aquí es *flexible*: no significa que los procesos no puedan continuar con otras labores, solo que el `agenteOrd` puede dar diversos usos a sus vectores de terminación. Por ejemplo, si se programa para tal efecto, el `agenteOrd` puede proceder, bajo presión de tiempo, con una siguiente tarea aún con procesos que no hayan reportado terminación.

4. Ejemplo de cómputo en paralelo con apoyo en dos esclavos

En aras de un ejemplo que ponga en práctica el concepto de vector y los de sublabores, y a realizarse en tres computadoras remotas, tendremos una tarea o labor que se asigna a un proceso ubicado en un nodo-e `nodoA` con nombre `oso@tosh`, y otros dos procesos con nodos-e como sigue: `nodoB` con nombre `osina@tosh` y `nodoC` con nombre `osona@tosh`. El programa que se cargará en todos los nodos es nombrado `merge.erl`. Para el `nodoA` tendremos un nodo llamado `oso` y la activación de nodo inicial con `merge:startA()`, quedando en activo un proceso en este nodo-e llamado `agenteOrd`, mismo que es controlado localmente por el usuario `operadorA`, con usuarios `operadorB` y `operadorC` para los demás nodos, con su correspondencia a cada nodo-e B y C .

Realizamos ahora las siguientes acciones para activar nodos-e y procesos participantes: en un nodo llamado `osina` ejecutamos `merge:startTrab(oso@tosh)` y en otro llamado `osona` ejecutamos `merge:startTrab(oso@tosh)`. Por medio del usuario `operadorB` interactúamos con el proceso `trabajador` definido sobre el `nodoB`, ejecutando `trabajador ! disponible`. Similarmente, por medio del usuario `operadorC` y el proceso `trabajador` definido sobre el `nodoC`, ejecutamos `trabajador ! disponible`.

Para este momento, el nodo-e que hospeda al proceso `agenteOrd` ya está preparado para realizar asignaciones de labores. Mostraremos esta parte por medio de una lista que deberá ser dividida en dos componentes, ordenada cada subcomponente aparte, y *cuando ambos subcomponentes estén listos*, las subsoluciones armadas darán una solución en total. Notemos que esto podría verse como una formulación del algoritmo de ordenamiento por fusión (*mergesort*) en la parte de armado total, con un trabajo ya sea intermedio o final de *fusión* o *intermezclado*, pero se deja en libertad a cada nodo-e auxiliar la manera de ordenar su parte: en este ejemplo, localmente se puede utilizar el algoritmo de ordenamiento rápido (*quicksort*), para enfatizar tal libertad.

La siguiente orden local que *agenteOrd* recibe por medio de su usuario `operadorA` es como sigue, dejando las demás partes en modo automático:

```
(oso@tosh)3> agenteOrd !
          {[3,2,4,5,2,1,2,3,7,8,9,10,2,6,8,2,3],comenzar}.
Comenzando cómputo en paralelo...
{[3,2,4,5,2,1,2,3,7,8,9,10,2,6,8,2,3],comenzar}
Trabajadores en activo...
        [{trabajador,osona@tosh},{trabajador,osina@tosh}]
```

```
Trabajador C terminó
Trabajador B terminó
Tareas terminadas
Resultado distribuido obtenido y armado:
        [1,2,2,2,2,2,3,3,3,4,5,6,7,8,8,9,10]
```

Por lo demás, el sistema aquí expresado en el estado actual sigue funcionando y nuevas solicitudes son posibles:

```
(oso@tosh)4> agenteOrd ! {[21,32,12,43,54,65],comenzar}.
Comenzando cómputo en paralelo...
{[21,32,12,43,54,65],comenzar}
Trabajadores en activo...
         [{trabajador,osona@tosh},{trabajador,osina@tosh}]
Trabajador C terminó
Trabajador B terminó
Tareas terminadas
Resultado distribuido obtenido y armado: [12,21,32,43,54,65]
```

En resumen, para solucionar este problema se ilustró con un par de agentes trabajadores. No obstante, con tres procesos trabajadores, y aún con una subdivisión irregular en el tamaño de las sublistas, puede perfectamente darse la ejecución concurrente ilustrada en las Figuras 1 y 2, donde el sentido de las flechas indica, de arriba hacia abajo, "se asigna", y de abajo hacia arriba, "se resuelve".

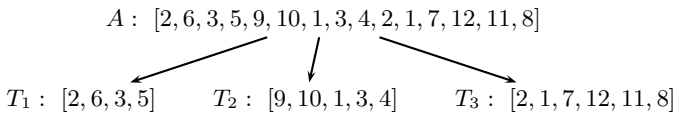

Fig. 1. Subdivisión que el agente A asigna los trabajadores T_1, T_2 y T_3.

4.1. Distribución de trabajo

En esta implementación notaremos que los trabajadores reciben sus órdenes de labor de forma vertical (ver Fig. 3), solo aceptando las directivas de trabajo sin ningún tipo de réplica; además, el conocimiento que tienen entre sí (a nivel de trabajadores) es nulo, así que esto hace imposible algún tipo de organización para enfrentar de manera horizontal las labores asignadas (coordinándose, posiblemente, entre sí). De darse la comunicación, esta puede darse, por un lado, mediante el `agenteOrd` como intermediario, o bien fungiendo como un servidor centralizado; esto puede orillar a que la terminación abrupta del agente

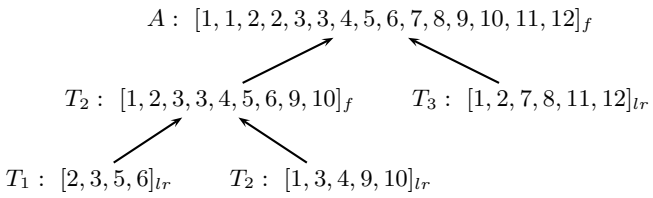

Fig. 2. Labores resueltas (lr), y ejecución del intermezclado (f). Note que el trabajador T_2 también está equipado para realizar intermezclado.

`agenteOrd` conlleve la terminación también abrupta de la realización distribuida de la labor en total; por otro lado, tomando un punto de vista más flexible, el `agenteOrd` podría terminar sin previo aviso, pero las subtareas asignadas todavía lograrse por medio de los trabajadores ya organizados y colaborando entre sí, ya sea surgiendo un trabajador como lider emergente o bien, aún sin ese liderazgo emergente, con las tareas ya finalizadas y almacenadas, para que cuando un nuevo agente lider surja sea notificado de qué se realizó mientras no había liderazgo "oficial". Si se utiliza tal almacenamiento, es necesario que la relevancia de la tarea completada (o lista por completarse por el nuevo `agenteOrd`) sea permanente en el tiempo.

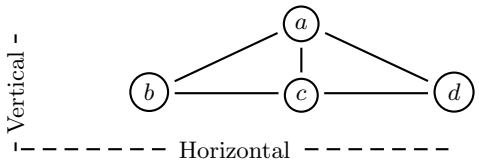

Fig. 3. Caracterizaciones de colaboración.

Ahora, el punto de vista horizontal se relaciona con *agentes colaborativos y cooperativos* [4], lo que por un lado debe formularse en un aspecto de paridad entre todos los agentes trabajadores, y por otro debe automatizarse en la posible reorganización de la labor aún sin los reportes finales, pero manteniendo un estado de resultado latente.

La primera parte para el conocimiento mutuo entre los agentes trabajadores es por medio de una solicitud del operador para conocer con nombre de referencia de nodos-e (ya que los procesos están identificados mediante `trabajador`) a quien tiene estos datos: el agente `agenteOrd`. Este conocimiento, cuando se haya autorizado, será solo de los agentes actualmente interconectados.

Procedemos como a continuación se indica:

Por la parte de un trabajador, quien inicia una solicitud:

```
solicitudConocer ->
        {agenteOrd,NodoR} ! {listaTrabajadores,node()},
        [reactivación de agente trabajador];
```

Por la parte del agente `agenteOrd`, quien procesa la solicitud del trabajador:

```
 {listaTrabajadores,Nodo} ->
            {trabajador,Nodo} ! {trabajadoresActuales,Ts},
                    [reactivación de agente agenteOrd];
```

Nuevamente, por la parte del trabajador quien recibe una respuesta a su solicitud:

```
{trabajadoresActuales,Ts} ->
io:format("Autorizado, trabajadores restantes:~p~n",[Ts]),
        [reactivación de agente trabajador];
```

Por ejemplo, suponiendo ya en funcionamiento un (pequeño) sistema distribuido con dos trabajadores además del agente `agenteOrd`:

```
(osito@RieGau)5> trabajador ! solicitudConocer.
solicitudConocer
Autorizado, trabajadores restantes:[{trabajador,osito@RieGau,
                                     ["1","Silvestre",29]},
                                    {trabajador,osita@RieGau,
                                     ["0","Piolin",19]}]
```

`map(fun(X) -> filtrar(X) end,Ts)` procedería a obtener `NTS=[{trabajador,osito@RieGau}, {trabajador,osita@RieGau}]` donde `filtrar({T,N,Ls}) -> {T,N}.` es una función de biproyección. En este caso desde el mismo proceso se tiene un conocimiento del propio nombre indirectamente, que bien podría servir para otro propósito distinto al de conocer los trabajadores (tal como darse a conocer en una *coalición* de `agenteOrd`s, posibilidad contemplada en el formalismo de SCEL). De momento es mejor eliminar el propio nombre: `Dest=eliminar({trabajador,node()},NTS)` lo que desde `osito@RieGau` obtenemos que `Dest=[{trabajador,osita@RieGau}]`, el nombre del otro trabajador, y simétricamente para `osita@RieGau`.

5. Colaboración vertical

La *colaboración vertical* que ahora estudiamos es un tipo de colaboración de jefe a subordinado. Para esta parte se establece que el proceso A existente en un nodo-e tenga un papel especial, que de momento llamamos de gestoría, y será nombrado como `gestor`. Ahora existirán dos procesos B y C con sendos nodos-e que tendrán un papel de realizar cierta clase de labores, y será nombrados cada uno de ellos `trabajador`, notando que no hay problema en la confusión de los procesos ya que los nodos-e determinan exactamente qué proceso es referido mediante la tupla `{trabajador, node()}`.

Mediante este esquema, es posible que los trabajadores acuerden realizar una labor mediante una distribución inteligente de trabajo. Este tipo de colaboración será nombrado *horizontal*. En ocasiones, solo algunas condiciones pueden ser cubiertas, tales como una distribución equitativa de trabajo (sin considerar las capacidades de los trabajadores), y se supone a todos los trabajadores con las mismas cualidades de labor, sin especificar tipos de trabajo. Esto ayuda a simplificar la repartición equitativa de trabajo sin caer en problemas complejos de calendarizaciones.

Para una posible aplicación, cada trabajador debería, además de su identidad, dar a conocer su capacidad mediante un número (unidades de labor) y su disponibilidad temporal (si no está realizando una labor ya o tiene un compromiso de hacerlo).

Con la función `partes/2` tendremos una repartición de trabajo válida solo para las labores son múltiplos exactos del número de trabajadores (ver Fig. 4); los trabajadores se ven beneficiados por tener una carga de trabajo equitativa, pero cuando las unidades de labor no son exactas, el programa tratará la parte residual, como un caso especial, asignado a ser gestionado por el agente `agenteOrd`.

```
asignacion(Ntrabajadores,Labor) ->
                                Len=length(Labor),
                                Div=Len div Ntrabajadores,
                                PartesExactas(Div,Labor).
```

Fig. 4. Una política parcialmente correcta y justa de distribución de labor.

Con la función `exactasYno/2` de la Fig. 5 la labor restante se reubica para el primer trabajador, aún siendo cercana al número de trabajadores (pero sin igualarla o sobrepasarla), lo que ejemplifica una política de distribución injusta; la formulación original de `partes/2`, por otro lado, solo aceptaba la división exacta del número de unidades de labor entre el número de trabajadores, o de otra manera conducía a un *crash* del agente `agenteOrd`, y aunque era correcta y justa cuando tal división era exacta, era incorrecta en la división inexacta (con residuo diferente de cero). En esta ocasión, el *crash* es evitable, pero con política injusta.

```
asignacion(Ntrabajadores,Labor) ->
                                Len=length(Labor),
                                Div=Len div Ntrabajadores,
                                exactasYno(Div,Labor).
```

Fig. 5. Una política injusta de distribución de labor.

```
asignacion(Ntrabajadores,Labor) ->
                                  Len=length(Labor),
                                  Div=Len div Ntrabajadores,
                                  distribuir(Div,Labor).
```

Fig. 6. Una política justa de distribución de labor.

Otra política más estaría dada por el cambio de `exactasYno/2` a `distribuir/2` (ver Fig. 6): ésta última función reparte equitativamente el excedente unidad por unidad entre los trabajadores participantes. Esta es una de las políticas que no requieren negociación implementadas, pero que son justas al repartir equitativamente el excedente de trabajo entre los trabajadores participantes. En esta ocasión, se reparte el excedente unidad por unidad entre los trabajadores participantes, repitiendo varias rondas si es necesario, hasta que se agotan todas las unidades del excedente. Algunos trabajadores pueden trabajar menos, pero definitivamente, no se sobreexplotará a uno solo ni tampoco se tendrán casos de terminación anormal (aunque hay que anotar que para todos los casos, la cantidad de labor debe ser al menos un múltiplo exacto mayor que el cuadrado de los trabajadores participantes, una característica de la implementación actual).

Para cuando se tengan tratamientos justos de excedentes se podrían considerar *rondas* que permitirían terminar el trabajo en mayor tiempo pero justamente (ver Fig. 7). Es necesario enfatizar que, en otros escenarios, la *capacidad* de labor de un trabajador podría tomarse en consideración al repartir las sublabores, lo que haría que, en una posible política, aquellos trabajadores que no tienen la suficiente capacidad, digamos k de ellos, sean exentos de la asignación de labores, y solo se considere a los $n - k$ restantes; esto originaría el problema de que obligaría a reasignar las sublabores a menos trabajadores, quizás presentándose nuevamente el problema de rebasar la capacidad de los parte de los $n - k$ trabajadores disponibles; bajo el mecanismo de rondas, suponiendo sea éste implementado, se puede encontrar el mínimo de entre las capacidades de todos los trabajadores para comenzar globalmente a realizar la labor asignada, pero a cambio de tardar varias rondas.

Consideremos por ejemplo a tres trabajadores con una labor de 12 unidades. Es posible una división justa: $(4, 4, 4)$ (en donde cada entrada del vector corresponde a una asignación del i-ésimo trabajador, $i = 1.,3$); otras injustas: $(12, 0, 0)$, $(6, 0, 0)$ y así.

Este sistema en términos de programación tiene otras características: ha sido especialmente diseñado para evitar generar nombres dinámicamente y puede tomar n (con $n \geq 1$) trabajadores para realizar n sublabores. Además, a propósito no se han utilizado algunas funciones de Erlang para el tratamiento de diccionarios o registros (*records*), pues las listas, como estructura de datos en la programación funcional, tienen una buena teoría transformacional de apoyo; el cambio a otras estructuras de datos estaría justificado cuando fuera necesario almacenar en una base de datos los registros de los (muchos) posibles trabajado-

```
asignacion(Ntrabajadores,Labor) ->
                                    Len=length(Labor),
                                    Div=Len div Ntrabajadores,
                                    distribuir(Div,Labor).
```

Fig. 7. Una política justa de distribución de labor.

res; llegado el caso, los datos a tratar como labor pueden ser enormes en cuanto a espacio y pueden requerir de accesos a un medio no volátil de memoria para su almacenamiento; si el procesamiento de las sublabores requiere tiempo variado, será deseable quizás solicitar "estados parciales" de parte del usuario `operadorA` para monitorear el curso global de cómputo. Otro tema dejado de lado es que hay una pobre tolerancia a fallas en el formato de las sublabores: si una parte de la sublabor no la puede elaborar un trabajador, éste debería realizar lo que sí puede y dejar aparte lo que no, quizás reportándolo.

En principio, estas limitantes son posibles de superar, aunque nuestro objetivo es (y será con los siguientes sistemas) ejemplificar los sistemas distribuidos con un núcleo básico de primitivas de Erlang, usando este núcleo básico de Erlang como una herramienta computacional adecuada para ilustrar aportaciones de índole distribuida.

5.1. Labor segmentada

Para los casos en que una labor pueda segmentarse, pongamos por ejemplo que tenemos 3 trabajadores, cada uno con capacidad 6 y una labor de 17 unidades. En un reparto equitativo, cada trabajador toma 6 unidades de labor, así que queda 1 unidad todavía por asignar. Esta asignación es dada directamente por el agente `agenteOrd`, y es solo por imposición (sin negociación de por medio); de la cadena de mando, esta encomienda es *vertical*. Órdenes que surgen de esta manera no son de ninguna manera puestas en duda por los trabajadores de más bajo nivel jerárquico. Dado el problema de ordenamiento que estamos elaborando, es posible que haya n trabajadores, así que algunas funciones de trabajo propias de `agenteOrd` deben generalizarse a tratar este caso general de tamaño n (uno por cada trabajador participante). Por ejemplo, `fusionGeneral/1` toma n listas ya ordenadas y genera una nueva lista ordenada:

```
fusionGeneral([]) -> [];
fusionGeneral([A]) -> A;
fusionGeneral([A,B|Ls]) -> C=fusionar(A,B),
fusionGeneral([C|Ls]).
```

(Una versión general con función de alto orde es la del módulo `lists`: `foldl`.)

Por medio de una imposición directa, el trabajo quedaría repartido como `trab1:7`, `trab2:6`, `trab3:6`. Esto es arbitrario y rompe la regla de no dar más labor a un trabajador de la permitida por su capacidad. La consecuencia es que `trab1` podría o no hacer la labor (y si la hace, podría ser motivo de falla de

operación), sin responsabilidad de su parte. Si, por otro lado, se respeta la regla de la capacidad, tenemos el reparto: `trab1:6`, `trab2:6`, `trab3:6`, quedando con una unidad para repartir posteriormente (en otros posibles turnos globales o *rounds*). Notemos que la labor devuelta por cada trabajador es válida, pero que debe abrirse otra ronda de labor, esta vez del tipo: `trab1:0`, `trab2:1`, `trab3:0`, aunque esto requiere guardar el resultado de la ronda previa (que ya está ordenado), y esperar a recibir el resultado nuevo (que en este caso, es fácil de realizar), para finalmente armar el resultado final:

```
(oso@tosh)4> mergeAg05:fusionGeneral([[1,2,3],[4,5,6],[-3,-2,-1]]).
[-3,-2,-1,1,2,3,4,5,6]
```

Por ahora, nos enfocaremos a generalizar también el vector de terminación. Para dos trabajadores, el vector debe estar en el estado `{1,1}` indicando que se ha terminado la labor. Mencionábamos que esto es anti-declarativo, pero es efectivo en esta etapa de implementación (en la parte de diseño no hay dificultad en indicar que los trabajadores emitan una "labor terminada" unánime para indicar que todo el trabajo asignado —globalmente— se ha llevado a cabo). Junto con este vector, otro más mantiene las tareas o por completar o ya completadas. A continuación mencionamos algunas otras formulaciones que se involucrarían en la cooperación para la repartición de labores, planeadas como trabajo a futuro.

5.2. Coordinación entre agentes

En un sistema distribuido donde los agentes tienen algunos aspectos de inteligencia, es clave mencionar el aspecto de la coordinación entre los agentes, de tal manera que tal coordinación favorezca el objetivo para el cual, en principio, el sistema está diseñado. Para el tema la coordinación para los agentes propuestos, existen varias alternativas a explorar, considerando también cuando se involucren *conflictos*. Veamos algunas opciones.

Ruptura de conflicto por azar. La ruptura de conflicto por azar es un sorteo de trabajo, realizado por los mismos trabajadores. Por ejemplo, todos los agentes pueden tener la misma posibilidad de ser elegidos para la unidad inicial de labor, y o bien, en otra ronda el que ya fue asignado con una unidad participa o no. No hay posibilidad de desacuerdo, ya que en principio todos acordarían estar conformes con los resultados obtenidos.

Ruptura de conflicto por altruismo. Para la ruptura de conflicto por altruismo, aquellos agentes que se vean con posibilidades de realizar una labor la aceptarán de inmediato, siendo inclusive varias posibles unidades; el componente no determinístico está presente, pero no causa ningún problema, ya que por definición los agentes candidatos estarían en una cola de espera, con algún tipo de asignación de orden; la asignación de orden implica que durante un intervalo de tiempo prudente el agente altruista todavía estará de acuerdo en aceptar la labor solicitada, pero no implica que realmente la labor se realice, dejando la terminación como parte de las características de la asincronía del sistema, o informando de partes realizadas, o reportando la terminación global de labor.

Ruptura de conflicto por votación. El caso de ruptura de conflicto por votación se inspira en la política democrática de algunas sociedades humanas. Es de tipo horizonal. Primero, debe surgir un conjunto de candidatos. Estos candidatos son seleccionados al azar o por otro agente o por auto-propuesta. Los candidatos, para empezar, aceptan su candidutra, y promocionan sus mejores atributos (como una función de beneficio social, *welfare function*). Surgen varias posibilidades de ordenamiento de preferencia del electorado. Finalmente, en varias posibles rondas de votaciones se decide quién será el candidato ganador, que en este caso es aquel que tenga el trabajo a realizar. De realizarse una implementación de algún sistema de votación (ya que hay varios, cfr. [20], cap. 12), sería deseable suponer una capacidad de procesamiento básico de cada trabajador, ya que es posible *arreglar* las rondas de votación para manipular los resultados finales, aunque hay evidencia de que esto tiene casos que caen en tratamientos difíciles NP [3]. No hemos implementado ningún tipo de votacion horizontal que logre una ruptura de conflicto.

Ruptura de confilicto por registro histórico y distribución justa posterior. Este tipo de ruptura de conflicto pasa de ser imposibilitad de resolverse horizontalmente a intentar ser resuelta verticalmente, y en este caso el agente `agenteOrd` mantiene una posición salomónica o justa. Por ejemplo, la ruptura de conflicto se puede dar por registro histórico y la distribución justa primero se basaría en observar qué ha hecho un agente. Si para tal agente no existe carga de trabajo previo o ha sido ligera, se le asigna una labor; de lo contrario, se considera otro agente, y así. Si todos tienen registros similares, se procede a romper esta simetría por azar, degenerando en un caso de ruptura de conflicto por azar.

Argumentaciones. Consideremos la situación ahora en donde un agente A de capacidad C recibe una asignación de D unidades de labor; si $D \leq C$, todo procedería sin problemas; la parte conflictiva viene cuando $D > C$. En una primera aproximación, el agente A puede rechazar de tajo la asignación, pero en otro cauce se podría tomar la siguiente decisión (de tipo vertical): De $D > C$, la cantidad $D - C$ es positiva, e irrealizable. Con una posibilidad de solventar aquella parte que el agente sí puede hacer, en lugar de rendirse ante la imposibilidad de la carga asignada puede hacer una parte y devolver la otra sin tocar. El agente A entonces recibe una labor Ds de tamaño de D unidades y devuelve $\{Cs, Ds - Cs\}$, en donde Cs es la labor que fue realizada y $Ds - Cs$ es la labor que fue imposible de realizar.

Alicientes o estímulos. Si fuera necesario cuantificar la proclividad de un agente se puede disponer de una entidad de valoración de labor parecida a la moneda que se utiliza en sociedades humanas, siendo una de las formas de "recompensar" la labor humana. Opcionalmente, podría ser que existan jerarquías de trabadores en donde el aliciente al llevar cabo algún tipo de labores sea escalar en tal jerarquía (recategorizaciones). Bajo ésta circunstancia, es posible también implementar *políticas de negociación*, de tal forma que se involucren conceptos de economía en la forma de ejecutarse el sistema distribuido propuesto.

La implementación de estas conductas inteligentes de colaboración es un tópico que al momento estamos abordando. Al momento, se tiene la convicción de que Erlang provee de un ambiente general de programación de temáticas en inteligencia artificial (dura) que es tan bueno como algunos lenguajes tradicionales en este tipo de aplicaciones, tales como Lisp, con la ventaja de tener ya incorporadas algunas primitivas robustas y escalables de concurrencia asíncronas, con un modelo teórico fiable de fondo. fiable

6. Conclusiones

En este artículo se han presentado algunos programas concurrentes y funcionales que están en el complemento práctico de algunas teorías de colaboración para la realización concurrente de labores.

Utilizando Erlang, se ha logrado ver la interrelación que existe entre la identificación de un conjunto de mensajes apropiados, su envío y recepción, y mediante esta comprensión se ha presentado una metodología basada en diálogos, mismos que fundamentan la colaboración y las políticas de repartición de trabajo. Nuestro problema a trabajar ha sido el de ordenar paralelamente una lista de números mediante un algoritmo de fusión, pero notemos que este problema puede no tener un tratamiento paralelo con otros algoritmos (por ejemplo, el de selección). Otros problemas, tales como los de grafos, pueden requerir de estrategias cuidadosamente planeadas para aprovechar la tecnología distribuida [14].

Frecuentemente en el área de la Inteligencia Artificial aparecen tareas que son exigentes en cuanto a cálculo computacional se refiere, o bien son por necesidad distribuidas. Ejemplos de la parte de exigencias de procesamiento serían: el reconocimiento de imágenes, la segmentación de videos, las redes neuronales y los algoritmos de geometría computacional; ejemplos de la parte distribuida serían los equipos de labores constituidos por varios robots y los agentes inteligentes de la Internet. El planteamiento de este escrito parte de un caso de estudio particular, ejemplificando las bondades de Erlang para el procesamiento distribuido de datos dentro del mismo lenguaje Erlang, pero se menciona como nota final que Erlang puede ser parte medular de un ecosistema donde diversos programas especialistas interactúan entre sí mediante la gestión de coordinación afianzada en los procesos de Erlang.

Agradecimientos. Gerardo Carreño agradece al Banco de México el apoyo económico otorgado para sus labores de investigación, y a la Universidad Tecnológica de la Mixteca por brindarle un agradable ambiente académico. Manuel Hernández agradece a la Universidad Tecnológica de la Mixteca las facilidades otorgadas para la realización de este trabajo. Particularmente, agradece a la Mtra. Mónica García, coordinadora de la Maestría Virtual de esta Casa de Estudios, su confianza y apoyo para obtener este artículo.

Referencias

1. Armstrong, J.: Programming Erlang: Software for a concurrent world. The pragmatic programmers (2007)
2. Armstrong, J.: Erlang. Communications of the ACM 53(9), 68–75 (2010)
3. Bartholdi, J.J., Tovey, C.A., Trick, M.: The computational difficulty of manipulating an election. Social choice and welfare 6, 227–241 (1989)
4. Bedrouni, A., Mittu, R., Boukhtouta, A., Berger, J.: Distributed Intelligent Systems, a Coordination Perspective. Springer (January 2009)
5. Bird, R., Wadler, P.: Introduction to Functional Programming. Series in Computer Science, Prentice Hall (1988)
6. Burstall, R.M., Darlington, J.: Some transformations for developing recursive programs. SIGPLAN Not. 10(6), 465–472 (Apr 1975), http://doi.acm.org/10.1145/390016.808470
7. Cesarini, F., Thompson, S.: Erlang programming. O'Reilly (2008)
8. Dunen-Keplicz, B., Verbrugge, R.: Teamwork in multiagent systems. A formal approach. Wiley series in agent technology, Wiley (2010)
9. Fokkink, W.: Modelling Distributed Systems. Springer (2007)
10. Fredlund, L.Å., Earle, C.B.: Model checking Erlang programs: the functional approach. In: ERLANG '06: Proc. of the 2006 ACM SIGPLAN workshop on Erlang. pp. 11–19. ACM (2006)
11. Fredlund, L.Å., Svensson, H.: McErlang: a model checker for a distributed functional programming language. In: ICFP '07: Proc. of the 12th ACM SIGPLAN Intl. Conf. on Functional programming. pp. 125–136. ACM (2007)
12. Huch, F.: Learning programming with Erlang. In: ERLANG '07: Proc. of the 2007 SIGPLAN workshop on ERLANG Workshop. pp. 93–99. ACM (2007)
13. Larson, J.: Erlang for concurrent programming. Queue 6(5), 18–23 (2008)
14. Lenharth, A., Nguyen, D., Pingali, K.: Parallel graph analytics. Commun. ACM 59(5), 78–87 (Apr 2016), http://doi.acm.org/10.1145/2901919
15. Noll, T., Roy, C.K.: Modeling erlang in the pi-calculus. In: ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang. pp. 72–77. ACM, New York, NY, USA (2005)
16. Partsch, H.: Specification and transformation of programs. Springer-Verlag (1991)
17. Pettorossi, A., Proietti, M.: Rules and strategies for transforming functional and logic programs. ACM Comput. Surv. 28(2), 360–414 (Jun 1996), http://doi.acm.org/10.1145/234528.234529
18. Varela, C., Abalde, C., Castro, L., Gulías, J.: On modelling agent systems with Erlang. In: ERLANG '04: Proc. of the 2004 ACM SIGPLAN workshop on Erlang. pp. 65–70. ACM (2004)
19. Varela, C.A.: Programming Distributed Computing Systems: A Foundational Approach. The MIT Press (2013)
20. Wooldridge, M.: An Introduction to MultiAgent Systems. Wiley Publishing, 2nd edn. (2009)