Simulador de sistemas P distribuido en WAN

Antonio Jiménez Martínez

Máster Universitario en Ingeniería Informática y de Telecomunicación

Major Computación Natural

Universidad Autónoma de Madrid

Escuela Politécnica Superior

Septiembre 2013

El resultado de este trabajo ha sido presentado en una charla invitada en la 2ª escuela de verano de la Red de computación natural celebrada en la UPM entre los días 23 y 25 de septiembre de este año (http://redbiocom.es/ISBBC/ISBBC13/Program.html).

1. Introducción

La computación bio-inspirada o computación natural constituye un nuevo campo en la informática en la actualidad muy activa desde el punto de vista de la investigación. Está comenzando a disponer de sus propios espacios de difusión de resultados (como por ejemplo la serie de conferencias bianuales IWINAC http://www.iwinac.uned.es/current/ y diversas redes de actividad nacionales http://users.dsic.upv.es/grupos/tlcc/rtematica/ e internacionales http://www.cs.duke.edu/~reif/BMC/BMCinternat/EuropeBMC/EMCC.html)
Suele entenderse o bien como la capacidad que tiene la naturaleza para computar o bien la capacidad que la naturaleza tiene para inspirar computación. Desde el primer punto de vista algunos investigadores están experimentando con la posibilidad de que sean directamente los sistemas vivos los que realicen computaciones y están analizando los mecanismos reales con los que la naturaleza procesa la información. Desde el segundo punto de vista se están definiendo dispositivos de cómputo abstracto tomando como inspiración los mecanismos mediante los que la naturaleza soluciona de manera eficiente problemas complejos, con frecuencia NP. Es este segundo punto de vista en el que se sitúa este trabajo y será el único que consideremos desde ahora en adelante.

Desde esta concepción, los objetivos de la computación natural serían la definición, la descripción formal, el análisis, la simulación y la programación de nuevos modelos de cómputo (habitualmente con el mismo poder expresivo que la máquina de Turing) inspirados en la naturaleza y que les hace particularmente adecuados para la simulación de sistemas complejos y su posible uso como arquitecturas alternativas a la de von Neumann. Como subrayaremos con más precisión más adelante, desde la propia fundación de la informática se ha estudiado la naturaleza desde este punto de vista. En la actualidad asistimos a un interés renovado por este enfoque. Este

nuevo interés tal vez venga motivado por la cercanía de los límites de miniaturización de los dispositivos electrónicos implementados con la tecnología actual y que ha venido siendo una de las mayores fuentes de incremento de la potencia de los ordenadores que conocemos. Algunos investigadores apuestan por otros modelos de cómputo para los que se pueda encontrar tecnología que mejore la potencia de los computadores que los implementen. La mayoría de esos modelos comparten ciertas características interesatnes como ser inherentemente paralelos. Estas condiciones hacen que se puedan desarrollar nuevos algoritmos para atacar problemas exigentes desde el punto de vista computacional y obtener mejoras en el rendimiento, al menos temporal, respecto a los que se podrían desarrollar para arquitecturas convencionales (von Neumann). Esto sugiere la posibilidad de, una vez que se encuentre la tecnología adecuada, obtener arquitecturas para ordenadores más potentes.

Alguno de los "computadores naturales" mejor conocidos son los sistemas de Lindenmayer (una clase de gramática de derivación paralela), autómatas celulares, computación inspirada en ADN, computación evolutiva, sistemas multiagente, redes neuronales, sistemas basados en membranas (o sistemas P) y redes de procesadores que evolucionan (o NEPs).

Este trabajo está enfocado a los sistemas P.

Hay al menos dos grandes áreas en las que estos nuevos modelos pueden resultar de utilidad: como nuevas arquitecturas para ordenadores alternativas a la de von Neumann y como herramientas de modelado para la simulación de sistemas complejos para los que las aproximaciones convencionales (habitualmente basados en ecuaciones diferenciales) son difíciles de manejar.

En cualquiera de los escenarios se necesitan los dos siguientes pasos:

- Diseñar la instancia particular del modelo que sea capaz de resolver la tarea tratada. Este paso sería el equivalente a programar el nuevo ordenador.
- Ejecutar el modelo.

Respecto al paso de ejecución, hay también dos alternativas: o se dispone del hardware específico para el modelo o se tiene que simular en una arquitectura convencional. Aunque ha habido algunos intentos para desarrollar hardware específico para la ejecución de sistemas P [14] realmente nos encontramos lejos de disponer de arquitecturas suficientemente generales para todas las familias de sistemas P. Por lo tanto, en este momento todos los desarrollos basados en sistemas P requieren en una fase u otra simulación.

Este trabajo está dedicado a presentar una nueva aproximación a la simulación de sistemas P que trata de superar algunas dificultades que hemos encontrado en los simuladores actuales manteniendo en la medida de lo posible un enfoque flexible y general.

A continuación describiremos con un poco más de detalle este modelo.

Los sistemas P y sus variantes forman un área de investigación conocida como computación basada en membranas. Se inspiran en las células biológicas, en su estructura, su funcionamiento interno y su interacción química. El modelo abstrae la manera en la que los compuestos reaccionan y cambian el estado interno de una célula y también en la manera en la que atraviesan las membranas celulares para influir en otras células. El paralelismo intrínseco de los sistemas P radica en estar compuestos de muchas células que evolucionan en paralelo y dentro de las cuales, además, las diferentes moléculas reaccionan a la vez. El modelo de sistemas P fue presentado por Paun y otros en [8] y [16].

Informalmente, un sistema P consiste en un número de membranas estructuradas de una manera jerárquica en el sentido de que unas contienen a otras. La jerarquía de los sistemas P es de árbol. De esta manera, la única membrana que no está incluida en otra se llama piel. El exterior de la piel se conoce como entorno. Al comienzo de la computación, las membranas contienen un número finito de moléculas, catalizadores y reglas. Las moléculas son símbolos básicos sobre los que se computará. Los catalizadores tienen el mismo papel que en la naturaleza ya que aceleran y posibilitan ciertas reacciones. Cada regla especifica cómo los símbolos actuales son sustituidos por otros. También pueden especificar que

atraviesen la membrana donde están e incluso que se disuelva. Cada molécula está representada por un símbolo. Como en la naturaleza, dentro de una membrana puede haber muchas copias de cada símbolo. El tipo de dato utilizado para representar las moléculas del interior de las membranas es el multiconjunto (mitad vector, mitad conjunto, del vector comparte la posibilidad de tener más de una copia de cada elemento por su posición, del conjunto la irrelevancia el orden de los elementos que contiene)

A continuación se muestra una representación basada en la teoría de conjuntos (cada membrana aparece en una forma que recuerda a un conjunto) de un sistema P muy sencillo, presente en la literatura y que calcula cuadrados de números naturales de manera aleatoria. Esta figura se ha obtenido de Wikipedia Commons.

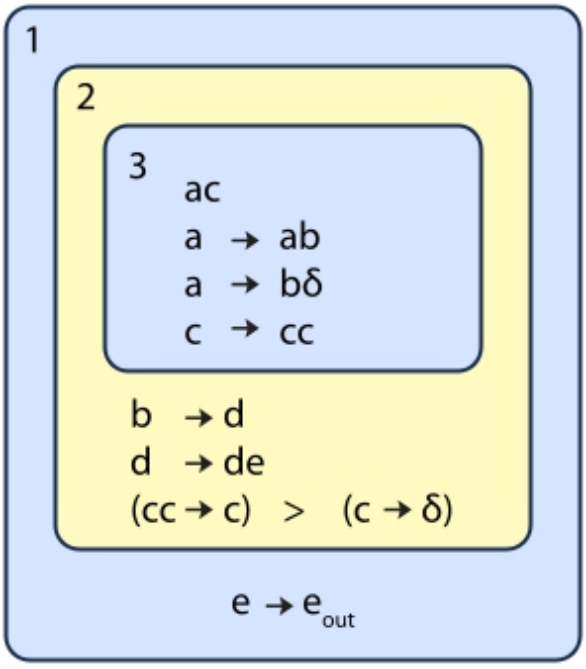

Imagen obtenida de: http://upload.wikimedia.org/wikipedia/commons/d/d6/P-system-example.svg

La computación se realiza a lo largo de un número discreto de intervalos de tiempo. En cada uno de ellos todas las membranas aplican sus reglas de manera paralela y no determinista. El paralelismo significa que cada posible aplicación de regla que sea posible tiene que ser efectivamente ejecutada.

El no determinismo significa que si el mismo símbolo o grupo de símbolos pueden ser utilizados por más de una regla, la elección se hace al azar.

A continuación se describe con más detalle la naturaleza de las reglas de los sistemas P.

Las reglas se representan de una manera muy habitual en la teoría de autómatas y lenguajes formales. Por A →B entendemos que las sustancias representadas por el multiconjunto A son necesarias para producir las que representa el multiconjunto B. Todos los símbolos consumidos son borrados de la membrana. Ya se ha explicado que si hay más de una regla para el mismo multiconjunto de la parte izquierda en principio se selecciona la regla que se aplica al azar. Esto puede ser modificado mediante la asignación de una prioridad a las reglas de manera que se ejecuta primero la más prioritaria. Basta con especificar un valor numérico que indica la prioridad de cada regla. Esto se ve en la aparición del símbolo > en la figura.

Respecto al ejemplo de la figura, es necesario mencionar que muestra el estado inicial del sistema P. El símbolo δ es un símbolo de disolución, su aparición dentro de una membrana se interpreta como una orden de desaparición. El propio símbolo desaparece y los contenidos de la membrana "caen" en su padre. Sus reglas se pierden.

Puede resultar interesante resumir una posible rama de ejecución de este sistema:

- Inicialmente el sistema sólo tiene dos símbolos en la membrana 3 ({a, c}) También pueden verse las reglas que contiene cada membrana.
- Sólo la membrana 3 funcionan ya que las otras están vacías. Supongamos que de todas las reglas disponibles para el símbolo a, se selecciona la primera (a→ab). Para el símbolo c se ejecuta la última que multiplica el número de símbolos c por 2. La membrana 3 tras este paso contendrá, por tanto abc^2.
- Supongamos que en este caso, para el símbolo a se selecciona la segunda regla (a→bδ). Su ejecución implica el borrado de la membrana y sus contenidos caen a la membrana 2 que acaba teniendo como contenido b^2c^4.

- En este momento la membrana 2 es la única no vacía. En este paso no puede aplicarse la segunda regla. Por su parte la regla b→d cambia todas las bs por ds. La regla cc→c es más prioritaria y divide entre 2 el número de símbolos c. Tras el paso ejecutado en este instante, esta membrana contiene $\{d^2c^2\}$
- No se puede aplicar en este momento la regla primera (no hay ya bs) la regla d→de añade símbolos e por cada símbolo d. De nuevo la regla cc→c divide el número de cs a la mitad. Tras ese paso el contenido de la membrana es $\{d^2e^2c\}$
- En el siguiente instante, de nuevo la regla d→de añade tantas es como ds hay. Sin embargo la regla cc→c ya no puede aplicarse porque sólo hay una c. Sin embargo si se puede ejecutar la regla c→δ que disuelve la membrana 2. Tras este paso, la membrana 1 contiene d^2e^4.
- En este instante, la regla $e \rightarrow e_{out}$ genera tantos símbolos e en el entorno como símbolos e hubiera en la membrana 1. En este caso se ha generado 4 (2^2)
- Al no haber posibilidad de cambiar nada durante una generación, el sistema para.

1.1. Objetivos

El objetivo de este TFM es implementar un simulador distribuido de sistemas P que intente evitar los inconvenientes detectados en otros simuladores, sobre todo en lo relacionado con el tamaño del problema que se puede solucionar y en el acceso a plataformas masivamente paralelas o distribuidas.

La idea es tener una red de nodos (ordenadores) dinámica, a la que se pueden unir nuevos nodos para realizar cálculos. Además es importante destacar que ninguno de los nodos conoce en su totalidad cómo está repartido el problema entre los nodos ni cómo están conectados los nodos. Para ello dejamos la parte de la comunicación en manos de una herramienta de mensajería, cuya eficacia está probada. En este caso la

herramienta utilizada es *RabbitMQ* (http://www.rabbitmq.com/), que es una implementación del estándar **A**dvanced **M**essaging **Q**ueuing **P**rotocol (http://www.amqp.org/).

1.2. Estado del arte

Entre los grupos de investigación interesados en las herramientas de desarrollo de aplicaciones para sistemas P, el Grupo de Computación Natural de la Universidad de Sevilla ha desarrollado P-Lingua, un lenguaje de especificación de sistemas P que realmente se ha convertido en el lenguaje de programación estándar (de hecho) para este tipo de ordenadores. Una de sus principales características es proporcionar un lenguaje de programación lo más cercano posible a la notación formal utilizada en la literatura. El objetivo es que los investigadores no tengan que hacer un esfuerzo suplementario para re-codificar utilizando una sintaxis nueva los sistemas P que hayan diseñado. Este mismo grupo también ha desarrollado los módulos Java necesarios para simular todos los sistemas P habitualmente presentes en la literatura. Pueden encontrarse más detalles en http://www.p-lingua.org y [15].

A pesar de que puede encontrarse la definición completa de la sintaxis en esa referencia la mostraremos a continuación comentando un ejemplo típico de la literatura que calcula cuadrados de números naturales de manera aleatoria.

```
@model<transition>
def main() {
        call n_cuadrados();
}
def n_cuadrados() {
        @mu = [[[]'3 []'4]'2]'1;
        @ms(3) = a,f;
        [a --> a,bp]'3;
        [a --> bp,@d]'3;
        [f --> f*2]'3;
        [bp --> b]'2;
        [b []'4 --> b [c]'4]'2;
        (1) [f*2 --> f ]'2;
```

```
    (2) [f --> a,d]'2;
}
```

- En la primera línea se especifica el tipo de sistema P (en este caso de transición)
- Pueden definirse módulos que pueden ser llamados desde otros (línea 3). En concreto main (línea 2) es el nombre usado para el módulo principal.
- La estructura se especifica mediante la palabra reservada @mu y las membranas representadas mediante corchetes. Cada membrana es seguida por su identificador.
- Los contenidos de una membrana se especifican mediante la palabra reservada @ms que contiene entre paréntesis el identificador de la membrana descrita.
- Las reglas se representan a continuación. En el caso de que se necesite especificar prioridades en las reglas, se especifica el orden entre paréntesis antes de la propia regla.

Algunos grupos de investigación han comenzado a considerar el uso de las llamadas GPUs [2] para la implementación de versiones específicas de sistemas P que se ejecuten de manera paralela. Hay que reconocer que mediante esta aproximación se pueden conseguir rendimientos temporales espectaculares tratando cada sistema P de manera adecuada. Nosotros preferiríamos ofrecer a la comunidad un mecanismo más general para el acceso a recursos masivamente paralelos.

Ha habido algunos intentos de simular de forma distribuida sistemas P. En [12] se describe un simulador distribuido de Java para la familia más sencilla y básica de sistemas P.

Estos investigadores utilizan el protocolo de Java RMI (http://docs.oracle.com/javase/6/docs/technotes/guides/rmi/index.html).En nuestra opinión, esa aproximación tiene algunas carencias asociadas con el proceso de acceso a cpus ubicadas en diferentes placas mediante redes (tanto clústers como redes de ordenadores) Una de las razones es que en el caso de los clústers requiere del uso de una máquina virtual de Java extendida de manera no estándar ya que la gestión de clústers no está

contemplada. Además, el protocolo RMI nos resulta una aproximación menos natural que otras opciones disponibles en lenguajes similares (como MPI en C++ [13])

Otros investigadores de nuestro grupo han explorado anteriormente otros enfoques en la simulación distribuida de sistemas P mediante el acceso masivo a recursos a través de internet. En [5] se utiliza por primera vez una aproximación basada en el paradigma map-reduce (http://es.wikipedia.org/wiki/MapReduce y [4]). Este trabajo puede considerarse como una prueba de viabilidad del enfoque. Se desarrolló un simulador para la familia de sistemas P de transición con creación y eliminación de reglas. En nuestra opinión el enfoque es prometedor. La tecnología utilizada sugiere su posible uso para acceder a cpus conectadas ya sea mediante internet o mediante un clúster. Una de las mayores dificultades (además de la necesidad de generalizar el tipo de sistemas P simulados) es la necesidad de que toda la información de una membrana tenga que ser guardada en el mismo nodo (cpu) Esto posibilita que en algún contexto pudiera fallar el sistema debido a la incapacidad del nodo que tuviera que almacenar esa membrana. Precisamente esta es una de las limitaciones que ha motivado el actual trabajo de fin de máster.

En los próximos párrafos se describe con cierto detalle esta aproximación. Nuestro interés es destacar las diferencias con el enfoque seguido en este trabajo de fin de máster.

El paradigma map-reduce fue desarrollado por los laboratorios de Google. Su propósito era mejorar la productividad en la solución de los problemas cotidianos asociados a su actividad al ocultar detalles de paralelización, tolerancia a fallos, distribución de datos y balanceado de carga. Es importante señalar que la mayoría de las necesidades de información de esta compañía implica la gestión de los datos como un conjunto homogéneo de registros cuya estructura se simplifica como parejas de clave-valor.

Un caso típico puede ser la acumulación (mediante una función suma, por ejemplo) de un campo de los registros de una base de datos relacional para sacar resultados por grupos (algo del tipo SELECT SUM(<campo que se suma>) GROUP BY <campo mediante el que se agrupa>)

Cualquier implementación de este paradigma incluye los siguientes pasos básicos:

- Una función Map que debe ser codificada por el programador para producir de cada conjunto de datos (parejas clave-valor) un conjunto derivado de nuevos pares. La idea es que este conjunto intermedio está más cerca de la solución del problema que el conjunto de partida. En el ejemplo del SUM-GROUP BY, esta función generaría, para cada registro de la base de datos, un par clave-valor con los campos a partir de cuyos valores se generarán los grupos como clave y cuyo valor sería el campo que se debe sumar.
- Una función Reduce que obtiene la solución del problema a partir de cada par intermedio. En el ejemplo que estamos explicando, esta función sumaría todos los valores recibidos para el mismo valor de clave (para el mismo grupo)

De esta forma, el programador sólo se concentra en estas dos funciones, con la única restricción de que los tipos en los pares clave-valor entre la función Map y Reduce sean compatibles. Hay que tener en cuenta que este paradigma está concebido para ser ejecutado en entornos distribuidos con múltiples máquinas (nodos) conectados. Eso implica que una aplicación desarrollada en este paradigma debe ser cuidadosa de los problemas de concurrencia-paralelismo que ella misma genere al margen del paradigma (por ejemplo si accede a recursos distribuidos o compartidos)

El paradigma sugiere algunas facilidades opcionales para minimizar los problemas que los desarrolladores puedan encontrar en el sentido expresado en los párrafos anteriores. Son las siguientes:

- *Counters*. Son contadores (por ejemplo de los registros generados) que se ofrecen gracias a que se calculan en el mismo proceso de *Map-reduce* sumando el número de registros en cada nodo (mediante un contador local) y luego acumulando ese valor.
- Funcion *combiner* para incrementar la eficiencia mediante una reducción local antes de que los nodos muestren al resto sus resultados. Por así decirlo, antes de ofrecer un gran número de

registros a la red para que se realice el reduce global, los nodos pueden hacer un reduce local previo

Hadoop es una bien conocida implementación Java del paradigma Map-Reduce pensada para ser ejecutada como una herramienta tipo *cloud.* Está libremente accesible en http://wiki.apache.org/hadoop/Distributions%20and%20Commercial%20Support que añade como ventaja adicional la posibilidad de depurar y monitorizar la ejecución de las aplicaciones que se estén desarrollando. Esta última característica es especialmente atractiva debido a la complejidad de la depuración en entornos distribuidos.

Hadoop incluye dos herramientas:

- Un sistema de ficheros distribuidos propio (**H**adoop **D**istributed **F**ile **S**ystem) y
- El motor que implementa el paradigma Map-reduce.

Y además está estructurado mediante dos tareas:

- Un maestro llamado *Job Tracker*
  - Que recibe las solicitudes de trabajo para MapReduce por parte de los clientes
  - Realiza tareas de recuperación del HDFS, mediante la re ejecución de las tareas de los clientes que fallen.
- Una tarea cliente en cada nodo llamada *Task Tracker*, responsables de la realización local del trabajo que cada nodo debe realizar dentro del paradigma MapReduce.

La filosofía de desarrollo de Hadoop ha mantenido la intuición y simplicidad del paradigma MapReduce, el programador sólo tiene que implementar una clase para cada función (Map, Reduce, Combiner) con el mismo significado y responsabilidades que en el paradigma MapReduce.

La simulación distribuida de sistemas P de transiciones mediante hadoop se basa en las siguientes decisiones:

- Cada ciclo de ejecución del paradigma MapReduce se hace coincidir con cada ciclo de evolución del sistema P.

- A cada membrana se le hace corresponder una llamada a la función Map. Potencialmente cada membrana podría estar contenida en un nodo diferente. Por esta razón, hay que llevar cuenta de un identificador global único para cada membrana. Esto no es necesario en otras aproximaciones de simulación de los sistemas P.
- Cada una de estas llamadas a Map simula el proceso de una membrana, esto es se producen los símbolos resultado de la computación consumiendo los que especifican las reglas. La función Map también emite los símbolos que deban abandonar la membrana utilizando para ello el identificador único global.
- A cada membrana, una vez terminada la fase Map, le corresponde una ejecución de la función Reduce que simplemente junta todos los símbolos de cada membrana y los coloca en ella haciendo uso del HFDS para realizarlo de manera trasparente.

En puntos anteriores hemos descrito brevemente PLingua, el lenguaje de especificación (o programación) de sistemas P que puede considerarse un estándar "de hecho". Para esta implementación distribuida hemos decidido extender PLingua, con los menores cambios posibles. Se ha decidido hacerlo de la siguiente manera:

- Cada membrana y todos los datos que contiene se guardan de manera independiente.
- Cada línea de los ficheros correspondientes a cada membrana tiene la siguiente estructura

GUID:label:structure:symbols

- Donde
    - GUID es el identificador único global.
    - Lable, es la etiqueta correspondiente a la membrana en el sistema P original (no distribuido)
    - Structure, contiene el primer nivel de membranas contenido en la descrita.
    - Puede contener también los símbolos propios de la membrana o bien estos pueden ser descritos al final (campo symbols)

A partir del siguiente fragmento de sistema P ejemplo descrito en PLingua

```
@mu = [[[]'3 []'4]'2]'1;
@ms(3) = a,f;
```

Podríamos escribir el siguiente fragmento de código PLingua para simulación distribuida:

```
101:1:[[]'2,102]'1
102:2:[[[]'3,103 []'4,104]'2]'101
103:3:[[a,f]'3]'102
104:4:[[]'4]'102
```

Respecto a las reglas, se sigue utilizando exactamente la misma representación de PLingua, no es necesario añadirlas en la línea que describe la membrana a la que pertenecen. El fichero que las contienen debe ser conocido por todos los nodos puesto que deben extraer de él las reglas que pueden aplicar.

El seudocódigo de las funciones analizadas anteriormente se muestra a continuación:

```
function Map(key, value):
        // create a small P sys, with the membrane context structure
        psystem = create_psystem(value)
        // simulates a step on the created system
        sim = create_simulator(psystem)
        stopped = sim.step()
        // locate membranes in the resultant structure
        parent = get_parent(sim)
        current = get_current(sim)
        children = get_children(sim)
        // emits the resultant structure to the related membranes
        if parent != null: emit(parent.id, parent.symbols)
        if current == null: emit(parent.id, current disolved)
        if current != null: emit(current.id, current)
        for children as child: emit(child.id, child.symbols)
        // increment counters to determine if the simulation is over
        mem_counter++
        if stopped: stop_counter++
function Reduce(key, values):
        parse values
        join structure + symbols received + dissolved children
        write to distributed filesystem
```

```
function Main():
        // run map and reduce while simulation is not over
        do
                mem_counter = stop_counter = 0
                configure enviroment
                execute map/reduce
        while mem_counter > stop_counter
```

1.3. Metodología y herramientas

A continuación enumeramos todo lo que hemos utilizado para el desarrollo del proyecto:

- Metodología:
  - Lean (se explica en la siguiente sección)
  - **T**est **D**riven **D**evelopment (se explica más adelante)
- Herramientas:
  - Gestión
    - IceScrum (http://www.icescrum.org/): gestión de las tareas del proyecto
  - Diseño y programación
    - IntelliJIDEA (http://www.jetbrains.com/idea/): usada para escribir el programa
    - Astah community (http://astah.net/): diseño del sistema y generación de diagramas
  - Compilación
    - Maven (http://maven.apache.org/): herramienta para compilar, generar código
  - Operativa del sistema
    - RabbitMQ (http://www.rabbitmq.com/): usado para la comunicación entre nodos del sistema

1.4. Lean Software Development

Es la adaptación al desarrollo software de la filosofía de producción de Toyota. Los esfuerzos de esta metodología se centran en eliminar el desperdicio [10].

El concepto es muy amplio y podríamos extendernos mucho, pero sólo presentaremos los conceptos más relevantes en nuestro caso (por ejemplo lo referente a equipos no es de interés en nuestro contexto).

En desarrollo software, el desperdicio puede aparecer en forma de:

- Cambio de tareas que se implementan
- Implementación de características que no son estrictamente necesarias
- Implementación de características que no generan ningún tipo de valor (incluso interno)
- Creación de deuda técnica
- Código que se deja a medias para retomarlo más tarde

Así, en nuestro caso hemos ido desarrollando lo que íbamos necesitando a cada momento. Con esto hemos evitamos el segundo y tercer punto de la lista anterior, y con ello eliminamos la necesidad de mantener código que no se va a utilizar y que no va a portar ningún tipo de valor. Además, lo hemos hecho teniendo en cuenta futuras necesidades. Es decir, no hemos creado deuda técnica, evitando el punto cuarto y así cerrarnos posibilidades en el futuro. Sin embargo, no hemos podido evitar del todo el quinto punto al no haber podido terminar una característica que teníamos planeada (pero está sugerido como mejora el terminarla).

Hay que tener cuidado y buscar cierto equilibrio, ya que no podemos gastar demasiado esfuerzo en hacer algo que valga para todo en el futuro, ya que podríamos equivocarnos en nuestra evaluación de "futuras necesidades" y en realidad estaríamos generando una complejidad extra que no es necesaria bajo ningún concepto, que no aportará valor en el futuro, y que habrá que mantener.

Esta situación puede venir producida porque tendemos a prever problemas. Y en lean hay una idea clara acerca de esto: un problema no es un problema hasta que aparece.

Para apoyar la toma de decisiones (como pueden ser las referentes al párrafo anterior), éstas han de tomarse lo más tarde posible, que es cuando más información se posee y cuando mejor se pueden tomar.

Aparte de todo lo anterior, es importante el no crear deuda técnica a base de introducir errores en el código. Y en caso de encontrar algún defecto (ya sea de diseño, funcional, o comportamiento inesperado), éste ha de ser subsanado lo antes posible.

Sin embargo, no hay que cometer el error de pensar que nuestro sistema puede estar completamente libre de fallos. Esto nos podría llevar a usar un cantidad excesiva de tiempo diseñando lo que nos llevaría a hacer over-engineering, que sería un desperdicio. Con esto entendemos que hay que aprender a vivir con algunos fallos, y aceptar que esto puede ocurrir.

Vemos cómo esta filosofía usa bastante el sentido común, ya que deja a cada uno decidir dónde está el equilibrio que necesita en varios de los puntos del desarrollo.

Veremos en la siguiente sección una metodología de programación que se adapta muy bien a esta filosofía: Test Driven Development.

1.5. Test Driven Development

Conocida por sus siglas, TDD, es una metodología de desarrollo software que se basa en la generación de código a partir de tests creados anteriormente a dicha generación. Al escribir primero los tests, éstos se pueden escribir pensando en la funcionalidad y por tanto representarán con fidelidad los requerimientos que el sistema a desarrollar ha de cumplir.

Así, el código que se escriba estará orientado a pasar dichos tests, y por tanto a cumplir los requerimientos y funcionalidad esperada, generando así

valor. Cómo se puede observar, es una forma casi natural de aplicar la filosofía *lean*.

Es un método iterativo, que contiene los siguientes pasos:

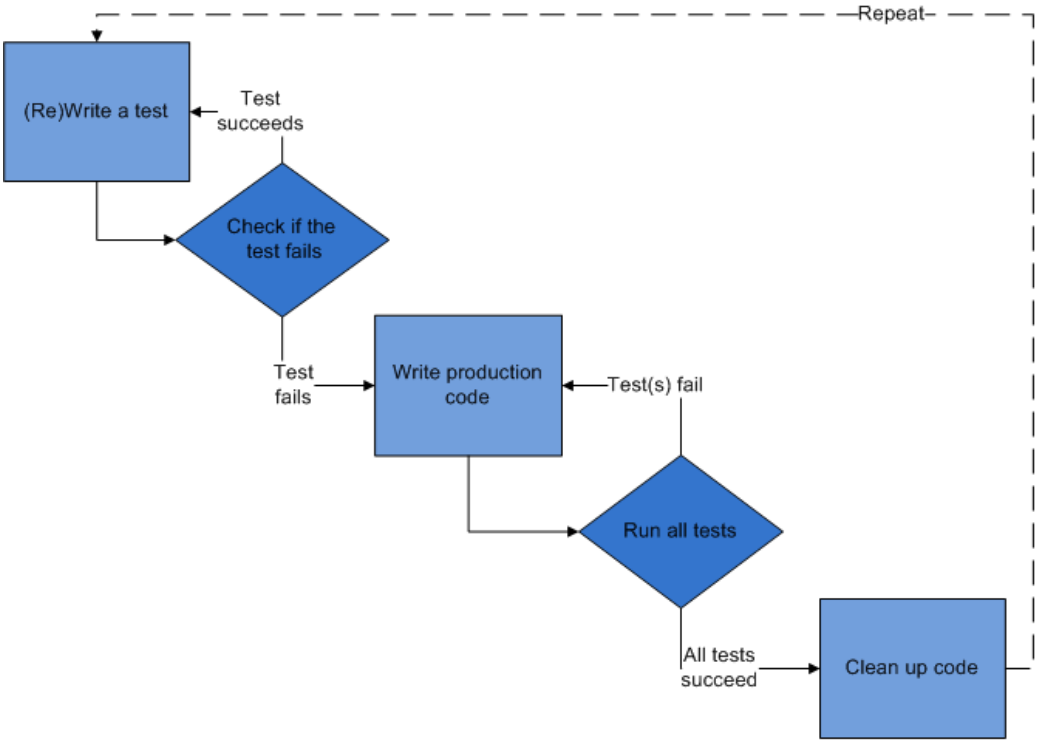

Imagen obtenida de: http://en.wikipedia.org/wiki/Test-driven_development

Es importante señalar los pasos tercero y quinto. En el tercer paso se aconseja que el código escrito sea el estrictamente necesario para pasar el test, por lo que puede no ser muy elegante y/o correcto desde el punto de vista de diseño y arquitectura. Es en el quinto paso donde el código se limpia y queda reutilizable y extensible. De otra forma se acabaría teniendo un código inmanejable.

Para la asimilación del método, y a veces con fines de reciclaje, esta práctica se lleva al límite en lo que se conoce como Extreme TDD. En esta modalidad se rompe la funcionalidad hasta tests extremadamente básicos, y el código escrito en el paso 3 puede llegar a resultar ridículamente simple. Sin embargo, esta forma de TDD no se utiliza para la producción de software sí.

NOTA:

En las siguientes secciones las posibles mejoras del sistema se irán encontrando poco a poco, y no agrupadas al final del documento como suele ser habitual. De esta forma conseguimos ponerlas en contexto, siendo así más fácil entender las razones por las que consideramos que pueden aportar valor al sistema.

La forma en la que mostramos las mejoras es mediante textos enmarcados. Los marcos son de distintos colores, cuyo código es:

Adición de funcionalidad o mejora de la implementación (nice-to-have features)

Aplicación de prácticas recomendadas o corrección bugs no importantes

Corrección de bugs importantes o adición de funcionalidad crítica para el funcionamiento a gran escala

2. Funcionalidad

En esta sección se presentará ***qué*** puede hacer el sistema. Una visión general de los bloques funcionales que lo componen es la siguiente.

Ilustración 1 – Bloques alto nivel

El diagrama anterior muestra las principales funcionalidades del sistema, y las relaciones entre ellas. Así, los pasos normales que se daría para diseñar y simular un sistema serían:

1. Creación de la definición del sistema (en el lenguaje creado para ello)
2. Simulación del sistema definido en el paso 1, que es la entrada para el simulador
   2.1. Conversión de la definición
   2.2. Simulación

Se podrán simular sistemas P de diferentes formas:

- En local
- Distribuido
  - Nodo maestro
  - Nodo proxy
  - Nodo esclavo

Dependiendo del modo de ejecución, el programa necesitará distintos parámetros, que se mostrarán al ejecutarlo sin argumentos.

2.1. Definición de sistema P y conversión

Este apartado funcional se compone de los elementos mostrados en el siguiente diagrama.

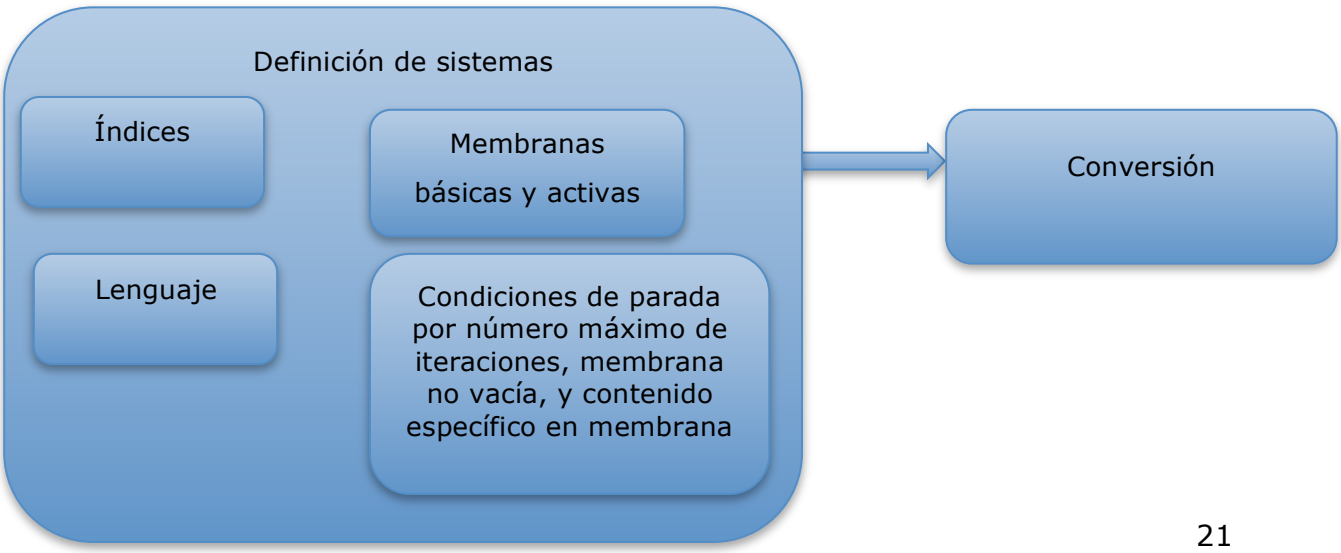

Ilustración 2 - Bloques funcionales de sistema de definición de sistemas P

Para aportar esta funcionalidad se ha diseñado un **lenguaje** basado en XML. Dicho lenguaje está definido por un fichero xsd (XML Schema Definition).

El hecho de utilizar xml y xsd hace que a la hora de escribir definiciones de sistemas P, y con el uso de las herramientas adecuadas, el usuario puede disfrutar de ventajas como autocompletado y validación sintáctica según va escribiendo, e incluso puede recibir sugerencias.

Sin embargo, estos lenguajes pueden no resultar cómodos para todo el mundo, por lo que se podría empezar por mejorar la experiencia del usuario desde este primer punto.

- Mejora del lenguaje:
  - hacerlo más fácil e intuitivo
  - eliminar elementos no utilizados (alfabeto)
- Creación de un VSDL (Visual Specific Domain Language), que facilite el acceso a la herramienta y su programación a expertos en los dominios susceptibles de ser simulados con ella.

El lenguaje propuesto soporta la definición y uso (en **índices** de símbolos) de **variables** con rango definido, e incluso combinación de las mismas a través de operaciones matemáticas.

Pero carece de otras características interesantes.

- Definición y uso de constantes
- Poder utilizar constantes y/o variables en la definición de otras variables
- Habilidad de no tener que definir variables hasta que no se vayan a utilizar (por ejemplo poderlas definir al utilizarlas como índice de un símbolo)

Con la ayuda de los índices, el usuario podrá definir sistemas con sus elementos típicos:

- Membranas iniciales → de dos tipos: **básicas, y activas**. Es interesante el uso de membranas activas ya que da más opciones en el diseño de sistemas P.
- Reglas → se ha diseñado el lenguaje y el sistema de tal forma que las reglas se puedan definir
- Condición de parada → 3 tipos:
    - **Número máximo de iteraciones** en la simulación
    - Que una **membrana** que cumpla las condiciones **definidas no esté vacía**
    - Que una **membrana** que cumpla las condiciones **definidas contenga una palabra determinada**

Una definición de un sistema en el lenguaje propuesto, puede ser utilizada como entrada del sistema para llevar a cabo su simulación.

Pero antes de ser utilizada por el simulador, esa definición ha de ser "traducida" a entidades que puedan ser manejadas por él, y esto se hace automáticamente en la aplicación.

Dicha traducción es una mera **conversión**, que en sí misma no conlleva análisis semántico, que es muy deseable desde el punto de vista de usabilidad. Por tanto el usuario sólo recibirá feedback de un proceso de análisis morfosintáctico de la definición: si la definición es errónea recibirá un mensaje de error. Es éste entonces otro punto a mejorar.

- Creación de un analizador morfológico

Actualmente la traducción soporta el uso de índices creando tantas entidades (reglas y/o símbolos) como diferentes valores de la combinación de los índices envueltos. Esto hace que la ejecución del sistema no sea tan efectiva como podría, por lo que es un punto a mejorar, pero hablaremos de ello más adelante ya que es una restricción del módulo de simulación.

2.2. Simulación local

Una parte del objetivo principal del sistema presentado es de simular sistemas P.

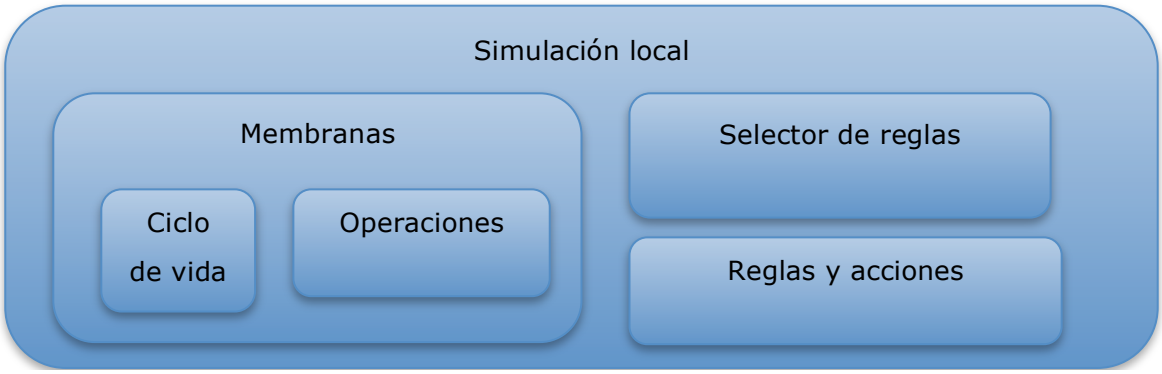

Ilustración 3 - Bloques funcionales del simulador en local

Para realizar esta tarea cuenta con un módulo con los elementos necesarios, que recibe como entrada una definición de un sistema P, y simula la ejecución del mismo.

Como se comentó en el apartado anterior, un sistema se define por los elementos característicos de los sistemas P:

- Alfabeto
- Conjunto de reglas
- Membranas iniciales
- Condición de parada

La simulación de nuestro sistema no requiere del alfabeto, pero hemos decidido incluirlo tanto en el lenguaje de definición como en los objetos que representan los sistemas P, para cumplir con la definición formal de un sistema P según la literatura, y para en un futuro aprovecharlo para la implementación del analizador morfológico mencionado en el apartado anterior.

La representación y el manejo de **membranas** se ha diseñado de manera que en un futuro se permitan deshacer **operaciones**.

- Permitir rollback de operaciones en membranas

La **selección** y aplicación de las **reglas** a aplicar se realiza siguiendo la lógica que se explica en la literatura. Una mejora que implicaría la extensión del concepto de sistema P, y por tanto del simulador.

- Permitir la mutación del sistema, es decir, la evolución de sus reglas

Sin embargo, aunque en la literatura hay algunos modelos que utilizan reglas con probabilidad, esta característica no está soportada por nuestro simulador.

- Soportar reglas con un atributo de probabilidad de que sean ejecutadas

2.3. Simulación distribuida

Es importante resaltar en esta sección la terminología utilizada:

- *Nodo* → ordenador que se participa en la simulación distribuida de un sistema P
- *Membrana* → compartimento de un sistema P, que puede contener símbolos y otras membranas

Esta funcionalidad es la que ofrece más valor. Permite la simulación distribuida en máquinas sin más restricciones que las que impongan las propias máquinas y las redes en las que éstas se encuentren.

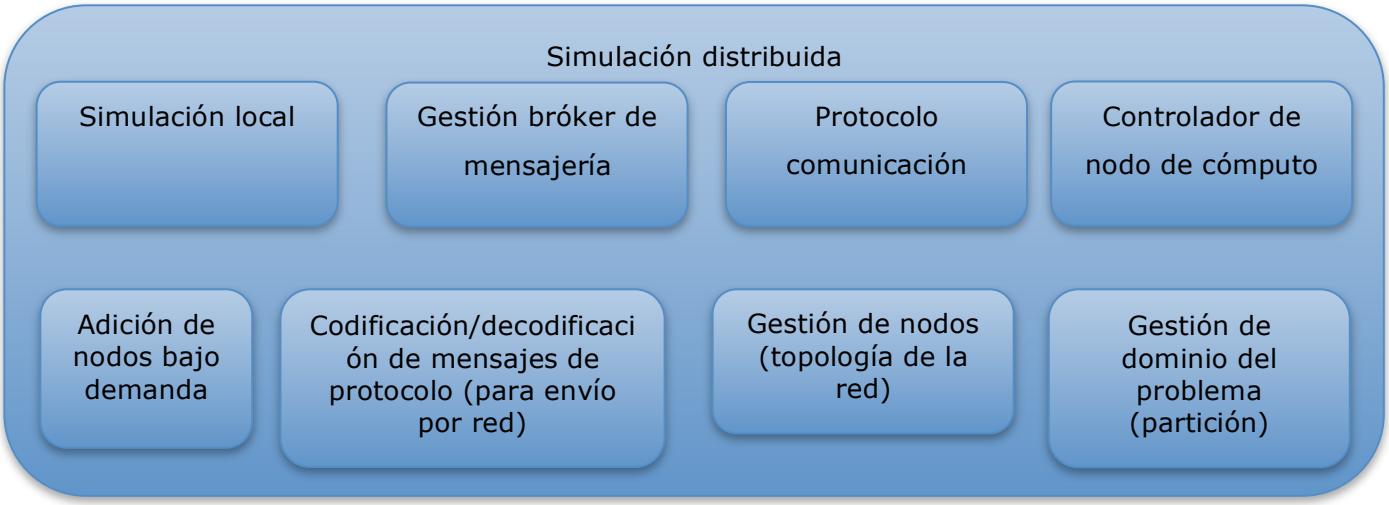

Ilustración 4 - Bloques funcionales simulador distribuido

Así, cualquiera desde su ordenador puede unirse a una simulación iniciada por un nodo al que llamamos ***maestro***.

Los nodos que se unen al cómputo pueden ser de dos tipos: ***proxy***, y ***esclavo***. Dependiendo de cual de los dos sean, tendrán unas u otras responsabilidades en el sistema, pero siempre con un objetivo común: la simulación del sistema.

Dicha simulación se hace de tal forma que se cumplen las siguientes premisas:

1. Las membranas del sistema P están distribuidas entre todos los nodos, y no hay ningún nodo que conozca la configuración de todo el sistema.
2. Un nodo sólo tiene consciencia de la existencia de sus nodos hijos, de su nodo padre, y del nodo maestro.

Con estas premisas se evita el problema de que un nodo se quede sin memoria cuando el problema es demasiado grande.

Esto acarrea el problema de la falta de robustez ante fallos en los nodos. Este problema no ha sido abordado en este sistema ya que aunque necesario para sistemas en producción, se quedaba fuera del ámbito del proyecto.

- Diseñar e implementar un sistema de replicado para soportar fallos en los nodos y así construir robustez en el sistema.

Cuando un nodo se une a la computación el sistema puede de buscarle un padre adecuado. Esta búsqueda de padre requiere de coordinación entre nodos ya que la premisa 2 no permite una búsqueda óptima de un nodo padre.

Asimismo, cuando un nodo se una a la computación, recibe una **partición del dominio** (un listado de membranas) para procesar. Este listado de membranas se selecciona en un proceso que está extremadamente ligado al de selección de padre.

Ambas operaciones han sido implementadas (en este caso inicial) para aprovechar el uso de RabbitMQ (bróker de mensajería AMQP) en los nodos maestro y proxies. El aprovechamiento de este elemento se hace consiguiendo tener un árbol lo más ancho posible.

Actualmente la entrada necesaria para la selección del nodo padre de un nuevo nodo consiste en información referente al número de membranas a procesar por los nodos hijo y al número membranas "piel" presentes en cada uno de esos nodos hijo. Esta información forma parte de un informe de estado que los nodos hijos comparten con sus respectivos padres. Sin duda hay otros parámetros interesantes para formar parte de esa información de estado.

- Recursos hardware del nodo
- Localización geográfica
- Número de membranas manejadas en el subárbol del que el nodo es padre (menos local que la información que actualmente se aporta)

A pesar de tener una estructura de árbol, y a pesar de existir algoritmos de balanceo e árboles, esta funcionalidad no se ha implementado, y aunque no es crítica para el correcto funcionamiento del sistema, sí que es deseable desde el punto de vista del rendimiento del mismo.

- Añadir habilidad de balanceo de árbol de nodos

3. Diseño e implementación

El sistema ha sido implementado en lenguaje Java, ya que es multiplataforma y ampliamente utilizado, lo que facilitará el ser extendido y mejorado.

Además, se ha diseñado de tal forma que sea extensible y se puedan aplicar distintos enfoques a las partes del sistema (colocación de nuevos nodos, partición de dominio, etc…). A tal fin, se ha hecho uso de interfaces y clases abstractas en todos los puntos del sistema que eran susceptibles de tener distintas soluciones, y que pueden ser relevantes en términos de rendimiento.

3.1. Simulador en local

A grandes rasgos, el simulador en sí (sin contar la definición del lenguaje y el conversor) está formado por los siguientes bloques.

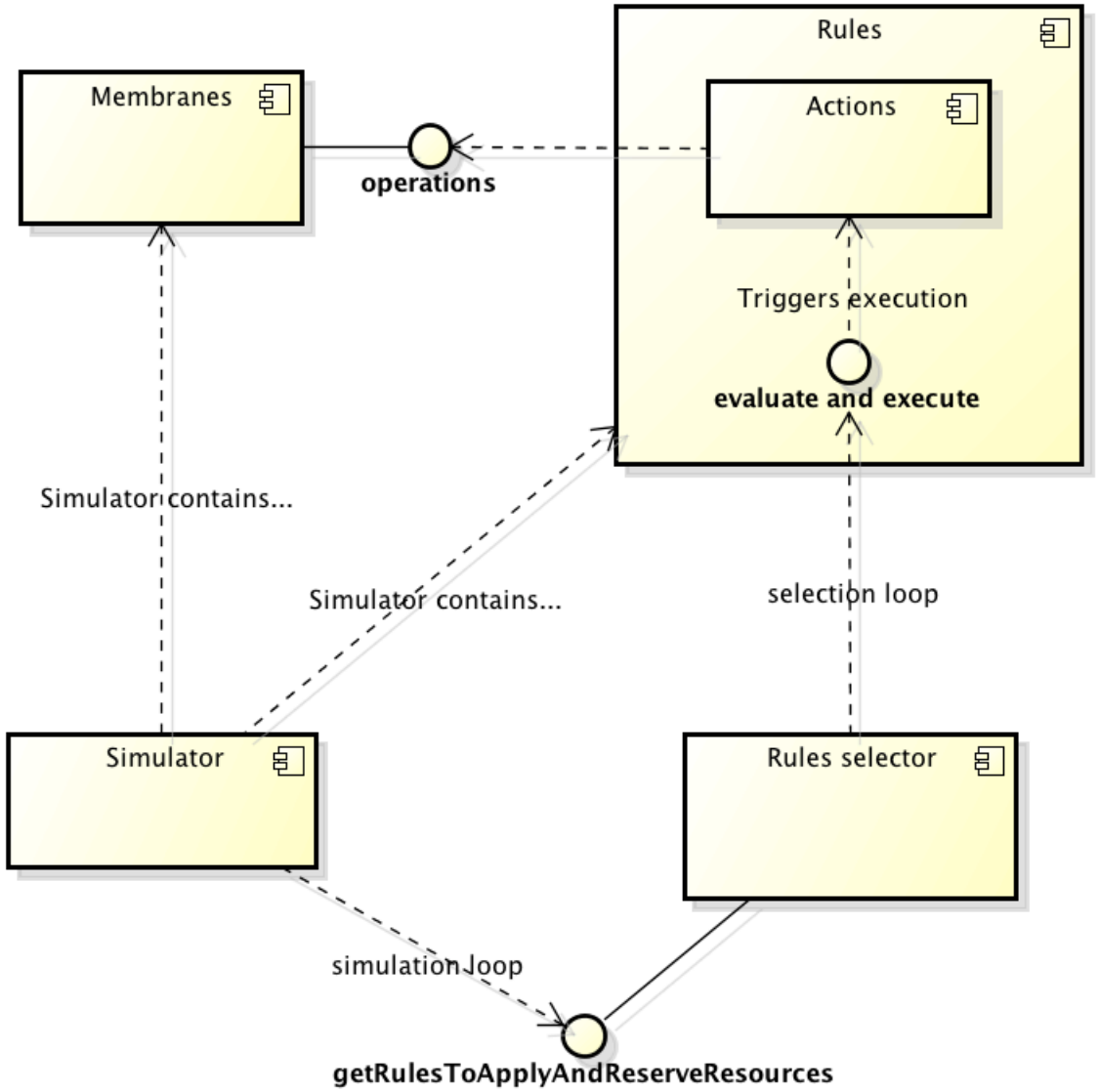

Ilustración 5 - Componentes simulador local

Proponemos la siguiente mejora aplicable a toda la aplicación.

- Mejorar la gestión de excepciones

3.1.1. Lenguaje y conversor

El **lenguaje** diseñado para generar ficheros de definición está especificado en un fichero xsd (XML Schema Definition). Ciertamente no es tan fácil o intuitivo como podría, pero conociendo el significado de las etiquetas y los

atributos y teniendo un IDE que ayude al proceso de creación de XMLs que cumplen con un xsd, se pueden definir gran cantidad de sistemas P.

Quizá la parte más compleja es la de definición de reglas, ya que no están separadas por tipos (sustitución, disolución, etc…). Actualmente consideramos las reglas como unos parámetros que serán finalmente convertidos a una serie de acciones que se agrupan en reglas.

Y aunque es sólo cuestión de diseño del lenguaje, ya que la función del conversor es hacer transparentes al simulador el lenguaje y sus peculiaridades, el dominio en cuestión (el de los sistemas P) no queda completamente reflejado con este enfoque.

- Cambiar la definición para que acepte diferentes tipos de reglas, y que éstas sean contenedores predefinidos de acciones

Esto es un ejemplo de definición de una regla:

```
<substitutionRule dissolves="false" priority="1">
      <environment charge="-" tag="e"/>
      <resultEnvironment tag="e" charge="+"/>
      <leftSide>
            <symbol symbol="q_2*k+1_" nReps="1"/>
      </leftSide>
</substitutionRule>
```

Se puede observar también en el trozo de código la utilización de índices en los símbolos: la subcadena entre los caracteres "_" del símbolo indica la presencia de un índice (con o sin operaciones).

En este caso, el índice *k*, que está definido (por un símbolo y acotado inferior y superiormente) al principio de la definición del sistema provocará que haya tantas reglas de este tipo como valores posibles del índice. Si por

ejemplo *k* va de 0 a 2 tendríamos 3 reglas con las siguientes partes izquierdas: q_1_, q_3_, q_5_.

Esta creación de reglas ocurre en el **conversor**. Dicho conversor traduce los objetos de la definición en los objetos manejados con el simulador. Los objetos provenientes de la definición son creados automáticamente por una librería de la que hacemos uso. Dicha librería actúa en la fase *generate-sources* de *maven*.

A pesar de tener los objetos ya creados, no queremos utilizarlos dentro del simulador, ya que estaríamos dependiendo de librerías externas para el correcto funcionamiento de todo el sistema, mientras que teniéndolos como objetos externos, un cambio en la forma de generar los objetos sólo afectaría al módulo de traducción.

3.1.2. Membranas

El sistema soporta **membranas** de dos tipos: básicas, y activas.

La diferencia entre estas membranas son los atributos que tienen. Mientras las membranas básicas se definen por una etiqueta, las activas tienen también una carga asociada que puede tener tres valores, a saber, *negativo*, *positivo*, o *neutro*.

Así, sólo las operaciones que necesitan de estos atributos serán diferentes en cada tipo de membrana. Esto lo hemos conseguido teniendo una clase abstracta membrana con dichos métodos declarados como abstractos.

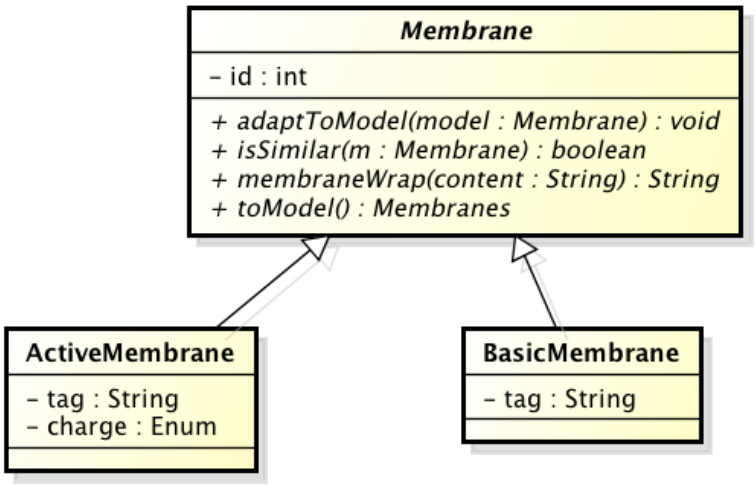

Ilustración 6 - Estructura de clases de membranas

La mayoría de la lógica está contenida pues en la clase abstracta, ya que es común a los dos tipos de membranas soportados.

Esta lógica la podríamos dividir en 2 partes, que usadas de forma coordinada permiten simular el comportamiento de una membrana:

- Operaciones realizadas en la membrana
- Ciclo de vida de la membrana

Las **operaciones** a las que hacemos referencia son todos tipo de manipulaciones más o menos complejas que se realizan en la membrana.

Éstas van desde añadir símbolos o membranas a una membrana, hasta dividir o disolverla. Es importante resaltar aquí que las operaciones más complejas hacen uso de las operaciones más sencillas. Así, el comportamiento de la membrana queda centralizado en estas operaciones básicas, las cuales son las de manejo de símbolos y membranas dentro la propia membrana.

Dichas operaciones básicas actúan sobre variables que guardan información dependiendo de la fase del **ciclo de vida** de la membrana. Esto lo conseguimos teniendo variables tipo *Map* en la clase *Membrane*, cuya clave es la fase del ciclo de vida, y el valor es la información que queremos mantener, que en nuestro caso es:

- Símbolos a añadir → procedentes de la ejecución de reglas tanto en el interior de la propia membrana como en la membrana padre o en alguna de las membranas hijas
- Símbolos a eliminar → por la ejecución de reglas en la propia membrana
- Membranas a añadir → procedentes de la ejecución de reglas tanto en el interior de la propia membrana como en la membrana padre o en alguna de las membranas hijas

Pero esto son las variables para hacer operaciones, y no indican el estado o configuración de una membrana. Ésta se mantiene con dos variables de tipo lista: una que contiene los símbolos, y otra las membranas hijas.

Así estos dos tipos de variables son utilizadas de la siguiente forma al ejecutar una regla de sustitución de símbolos:

1. Eliminamos los el precedente de la regla de la variable que contiene los símbolos que forman la configuración de la membrana
2. Añadimos dichos símbolos a la lista de símbolos a eliminar durante la fase X
3. Añadimos los símbolos del consecuente de la regla a la lista de símbolos a añadir durante la fase Y.
4. Al llegar la fase X limpiamos la lista de símbolos a eliminar durante esa fase
5. Al llegar la fase Y borramos la lista de símbolos a añadir durante esa fase y los añadimos a la lista de símbolos que realmente están contenidos en la membrana

El paso 2 (y la variable de estado temporal envuelta) y por tanto el 4 podrían parecer innecesarios, y además contravienen el principio del desarrollo *lean*, pero hemos decidido incluirlos ya que mantenemos la trazabilidad de lo que acurre en la membrana en cada momento, y puede ser un primer paso hacia la implementación de rollback de operaciones (esta mejora ha sido mencionada con anterioridad en el apartado de funcionalidad).

El siguiente diagrama muestra las diferentes fases del ciclo de vida de una membrana.

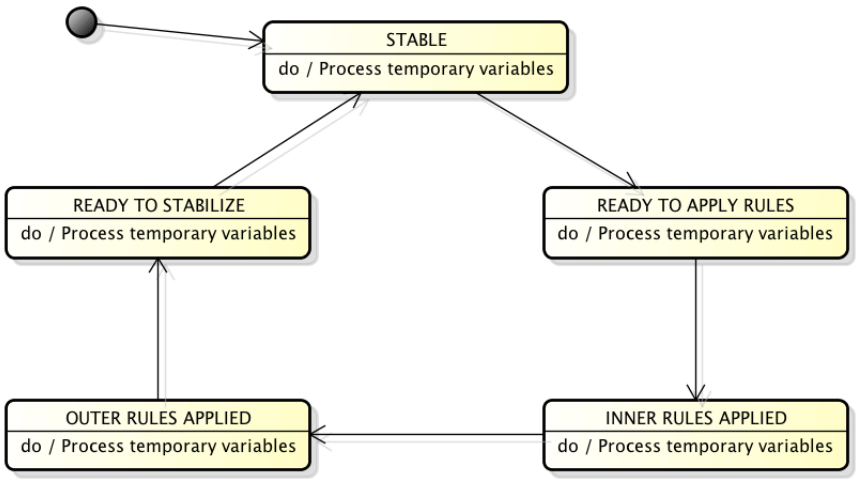

Ilustración 7 - Ciclo de vida de una membrana

Si bien no todas están en uso actualmente, no violamos la filosofía *lean* ya que los estados no necesitan un manejo especial para cada uno de ellos (excepto en el caso de la fase *STABLE* que se menciona explícitamente en el código).

3.1.3. Reglas

En nuestro simulador, las **reglas** son un simple contenedor de **acciones** que se aplican a las membranas que cumplen con las características requeridas por dichas reglas. Estas acciones son lo más básicas posibles, y hacen uso de las operaciones del objeto *Membrane*.

El siguiente diagrama muestra cómo están estructuradas las reglas y las acciones.

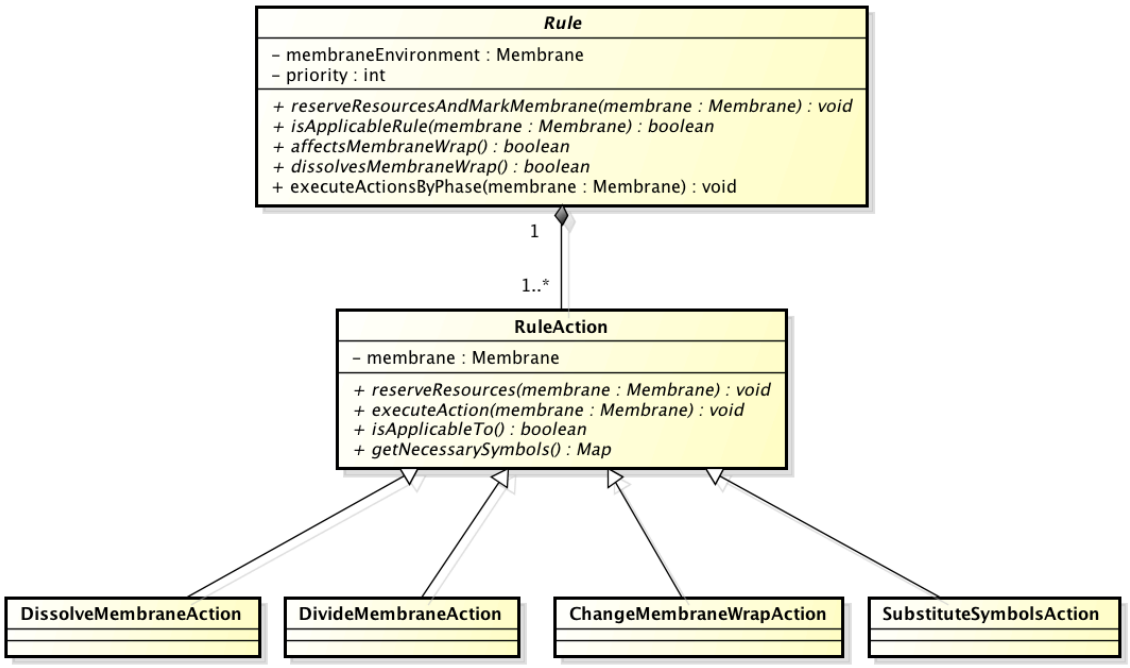

Ilustración 8 - Estructura de clases para reglas

Como se puede ver en el diagrama, las reglas tienen un atributo de prioridad, pero carecen de uno de probabilidad (se mencionó esta carencia en el apartado de funcionalidad).

En este diagrama sólo mostramos las clases relevantes, y hemos dejado fuera las clases a extinguir.

- Refactorizar para limpiar el motor de reglas de clases no utilizadas

La ejecución de estas reglas se realiza de manera maximal, es decir, una vez que una regla es seleccionada para aplicarse en una membrana, ésta se aplica tantas veces como su precedente se encuentre en dicha membrana. Esto es así excepto en los casos en que una regla modifica, divide, o disuelve dicha membrana.

Cuando una regla se evalúa como aplicable, su aplicación sigue los siguientes pasos:

1. Reserva de símbolos del precedente de la regla (de manera maximal si procede), y marcado de membrana para operaciones especiales (disolución, modificación, o división)
2. Adición de símbolos de consecuente de la regla
3. Aplicación de las operaciones especiales mencionadas en el paso 1

Observar que la adición y reserva (y eliminación) de símbolos de una membrana se llevan a cabo haciendo uso de las variables temporales asociadas al ciclo de vida de la membrana.

Si bien la evaluación de si una regla es aplicable en una membrana depende de la propia regla, la lógica de qué reglas evaluar y **selección de reglas** a ejecutar se hace en una clase implementada sólo con ese fin. Dicha clase sigue una clara estrategia, que para ser modificada requeriría de un cambio en la estructura de dichas clase y la refactorización del sistema.

- Hacer una interface para la selección de reglas, y así hacer más extensible el sistema.

Dicha selección se lleva a cabo en un bucle que comprueba las reglas por orden de prioridad, y dentro de una misma prioridad las comprueba de forma aleatoria. Y esto se hace para cada membrana del sistema. Es entonces cuando la estrategia de manejo de índices en los símbolos se antoja contraproducente, ya que generamos una gran cantidad de reglas cuando podríamos tener sólo una que comprobar.

- Soportar símbolos como objetos con un atributo para manejo de índices

3.2. Simulador distribuido

Como mencionamos antes, esta simulación se realiza repartiendo el problema entre los nodos que se vayan agregando a la red. Esta red de nodos es en realidad un ***árbol***. Podría ser deseable en un futuro generalizar esta topología y no restringirnos a tener un árbol, y para ello ya hemos pensado en varias mejoras en esa dirección.

- Permitir que un nodo pueda tener más de un padre.

Los nodos que conforman dicho árbol serán de 3 tipos: maestro, proxy, y esclavo.

El nodo maestro es el que inicia la computación, y es el único de este tipo durante una simulación.

Los otros dos tipos de nodos pueden aparecer en la red cumpliendo con la siguiente restricción: los nodos proxy pueden tener hijos, mientras que los esclavos no (son las hojas del árbol).

Tanto el nodo master como los nodos proxy han de tener instalado un bróker de mensajería con el que se comunicarán con sus nodos hijo. De

esta manera, una configuración en un momento dado de la simulación puede ser la siguiente.

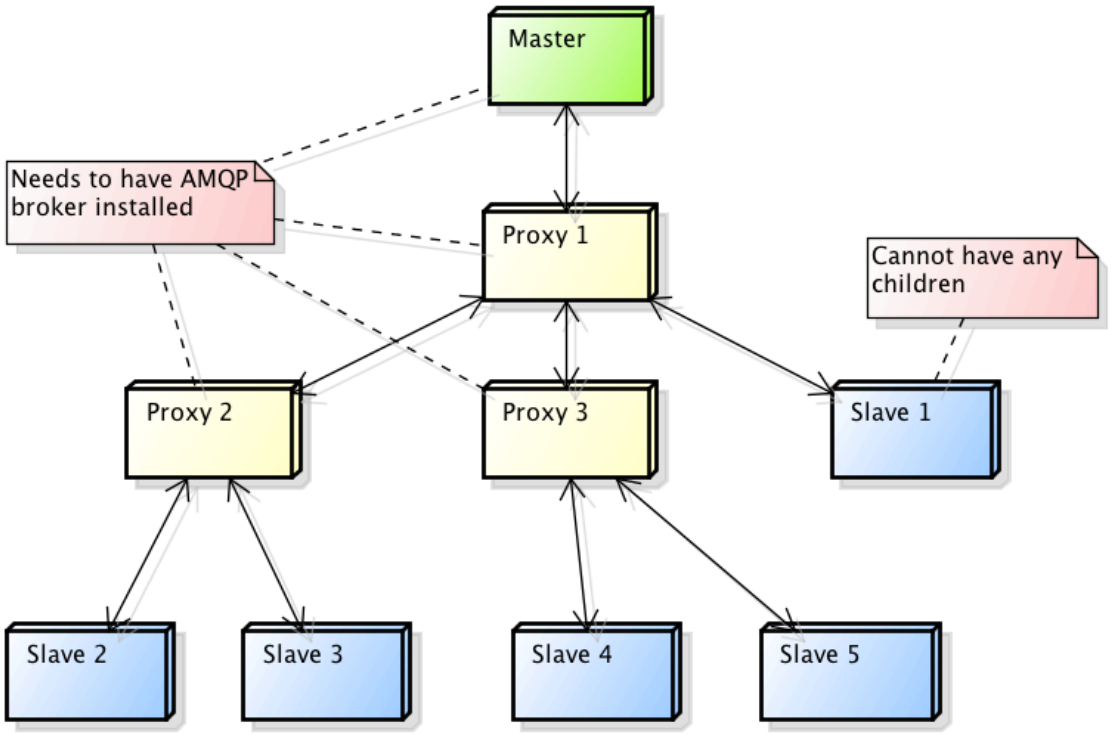

Ilustración 9 - Snapshot de un despliegue típico

Cada nodo contendrá una parte del dominio (del problema, de las membranas envueltas en la simulación) y una imagen de su entorno: conocimiento básico acerca de su padre y de cada uno de sus hijos.

Este conoci miento básico acerca de un nodo se compone de información para poder contactar con dicho nodo, y en el caso de los hijos también incluye información de estado (actualmente, pero es extensible) como carga de trabajo (número de membranas que tiene a su cargo), y formas en las que puede particionar su dominio. Mejoras a este respecto se han propuesto en el apartado dedicado a la simulación distribuida en la sección de funcionalidad.

- Mejorar separación en la definición de un nodo (ahora mismo identificador = ip + puerto):
  - Ip + puerto
  - Identificador

Por supuesto, todos los nodos tendrán la misma definición del sistema, y un motor para ejecutar el modelo sobre la parte del problema con el que han de trabajar. La aplicación que ejecuta un nodo está formado por los siguientes componentes.

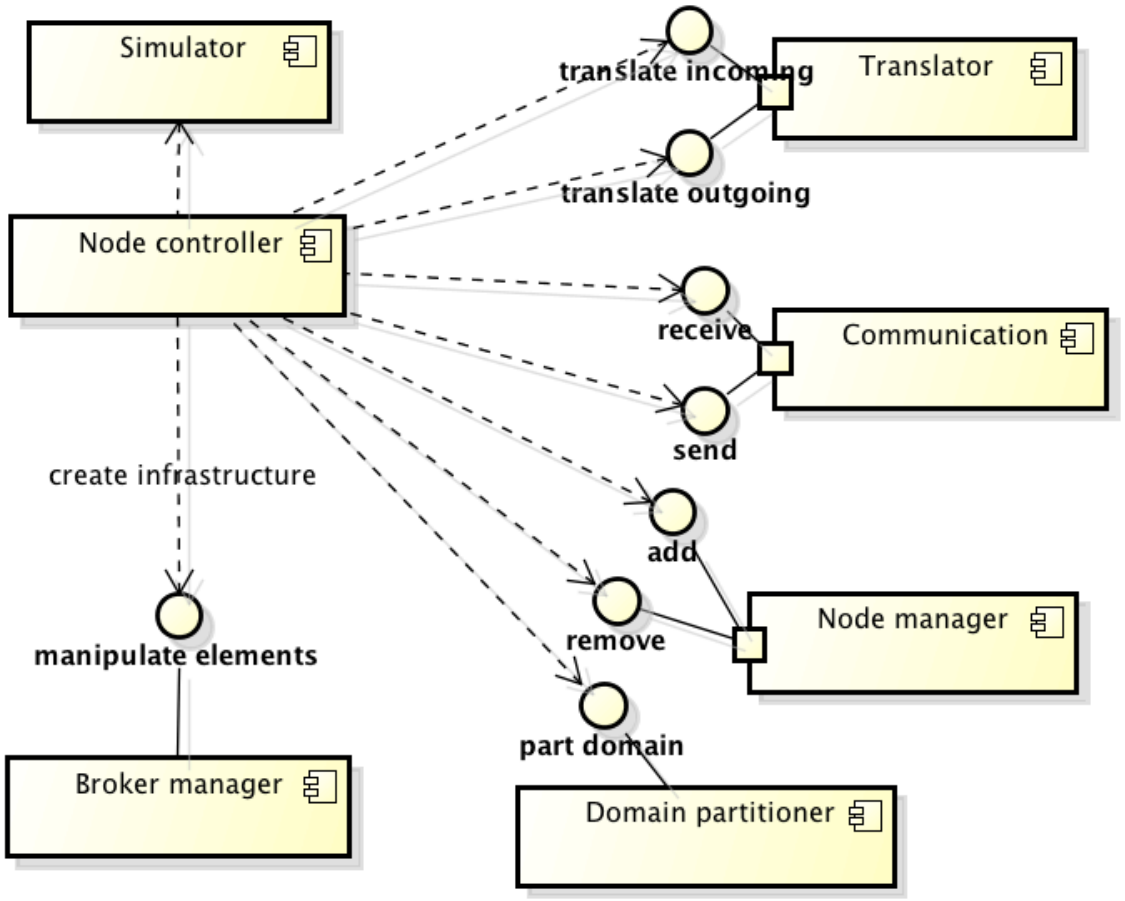

Ilustración 10 - Componentes del simulador distribuido

A grandes rasgos se puede observar que el comportamiento de la simulación distribuida está regida por el controlador.

Un controlador actúa de una manera bastante básica: espera mensajes (tanto de sus padres como de sus hijos) y actúa en consecuencia dependiendo del tipo de mensaje que reciba. Los distintos tipos de mensaje están definidos en el protocolo del sistema, que se verá más adelante.

Cada tipo de nodo actúa de manera ligeramente diferente, y utiliza los componentes del diagrama de más arriba de forma distinta.

3.2.1. Arquitectura de infraestructuras

Hemos mencionado anteriormente que cierto tipo de nodos (proxy y maestro) tienen el requerimiento de tener instalado un bróker de mensajería.

- Permitir a un nodo ser proxy haciendo uso del bróker de otro nodo.
- Nombres dinámicos de los *exchanges* y *queues*.

La razón de necesitarlo es proveer a un nodo de la habilidad de comunicarse con sus hijos y de recibir peticiones de las que no tenga consciencia (peticiones de otro nodo para aceptarle como hijo, nodos que se quieren unir a la computación, etc…).

Sin embargo son los nodos hijo los encargados de crear en el bróker de mensajería del nodo padre la infraestructura de colas por las que enviará y recibirá los mensajes que le permitirán comunicarse con dicho nodo padre.

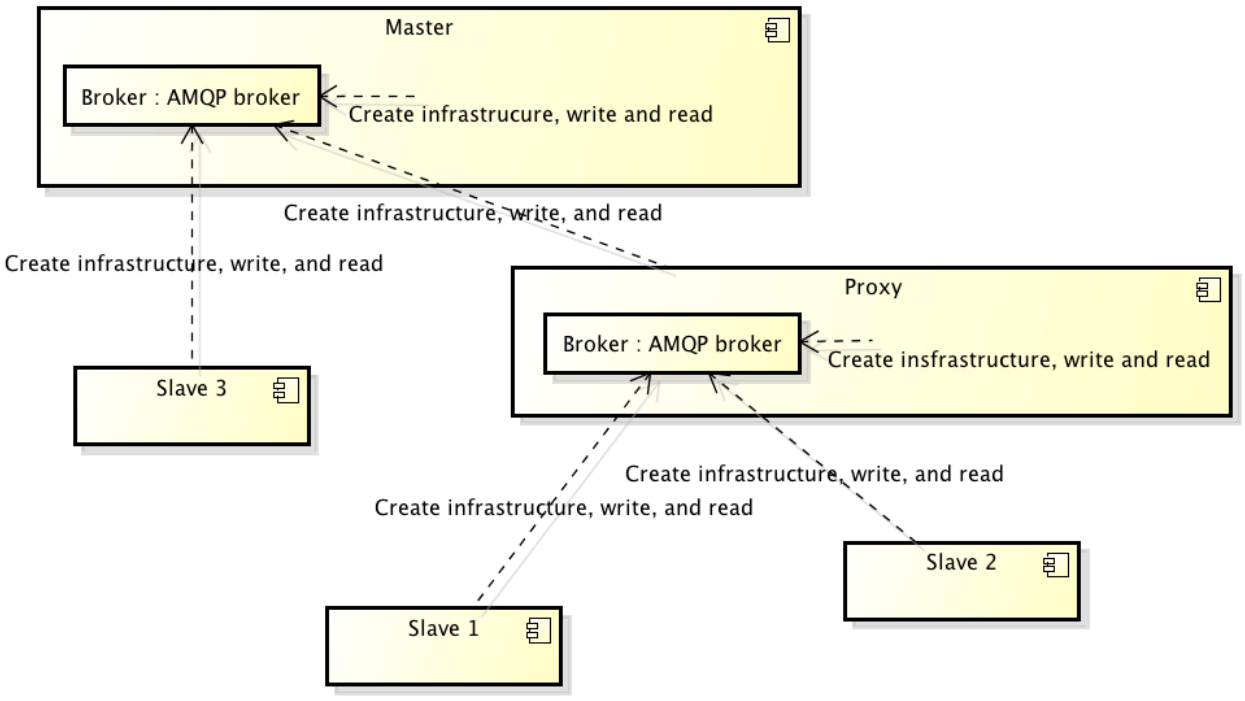

Ilustración 11 - Responsabilidades en la creación de colas y exchanges en los brokers de una configuración típica de nodos

Vemos así un patrón claro: cada nodo es responsable de crear la infraestructura necesaria para utilizar un bróker, aunque éste no esté en el mismo nodo.

Las infraestructuras de las que hablamos están compuestas de 3 tipos de elementos propios del sistema de mensajería:

- Exchanges
- Queues
- Bindings

Un símil para entender el rol de cada uno de estos elementos es el siguiente, y está basado en el reparto por correo ordinario de publicaciones a las que los usuarios están subscritos. El *exchange* sería la oficina de correos, que es la que se encarga de repartir las revistas/periódicos. Una *queue* es el buzón de usuario. Y un *binding* indica la revista a la que un usuario de esa oficina de correos está subscrito. Entonces, un *mensaje* será el paquete, que contendrá un *routing key* que en este caso será la revista con su nombre, y de esta forma la oficina de correo sabrá a qué usuarios repartir qué paquetes. Así, el diagrama siguiente muestra una situación típica.

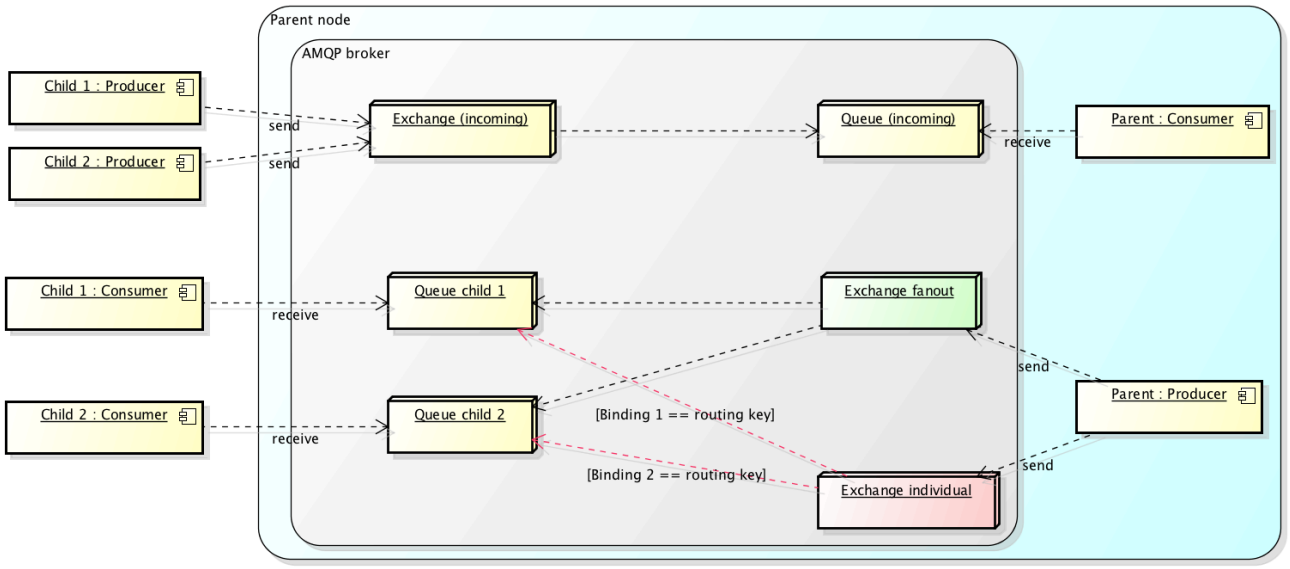

Ilustración 12 - Arquitectura de colas y exchanges

En ciertos casos, el *exchange* reparte mensajes a todas las *queues* conectadas a él. En ese caso no hay *binding* asociado.

Usamos entonces los exchanges con bindings para enviar mensajes dirigidos a un nodo hijo específico, y los exchanges sin bindings (más conocidos como *fanout*) para mensajes en broadcast hacia los nodos hijo.

Esta estructura de *exchanges* y colas es sencilla y funciona, pero podría mejorarse.

- Estudiar si una arquitectura de colas en la que separemos los mensajes de entrada por tipos de operación puede mejorar para el rendimiento del sistema.

3.2.2. Nodos

Aquí veremos en profundidad cómo actúa cada tipo de nodo para llevar a cabo su tarea.

En líneas generales se podría decir que el comportamiento de los nodos está regido en base a los mensajes que recibe de otros nodos. Es de este modo como la simulación permanece coordinada. Estos mensajes pertenecen a un protocolo, que se explicará más adelante.

Así, este es el esquema básico del funcionamiento de un nodo cualquiera (exceptuando el nodo maestro que no tiene que unirse a la computación).

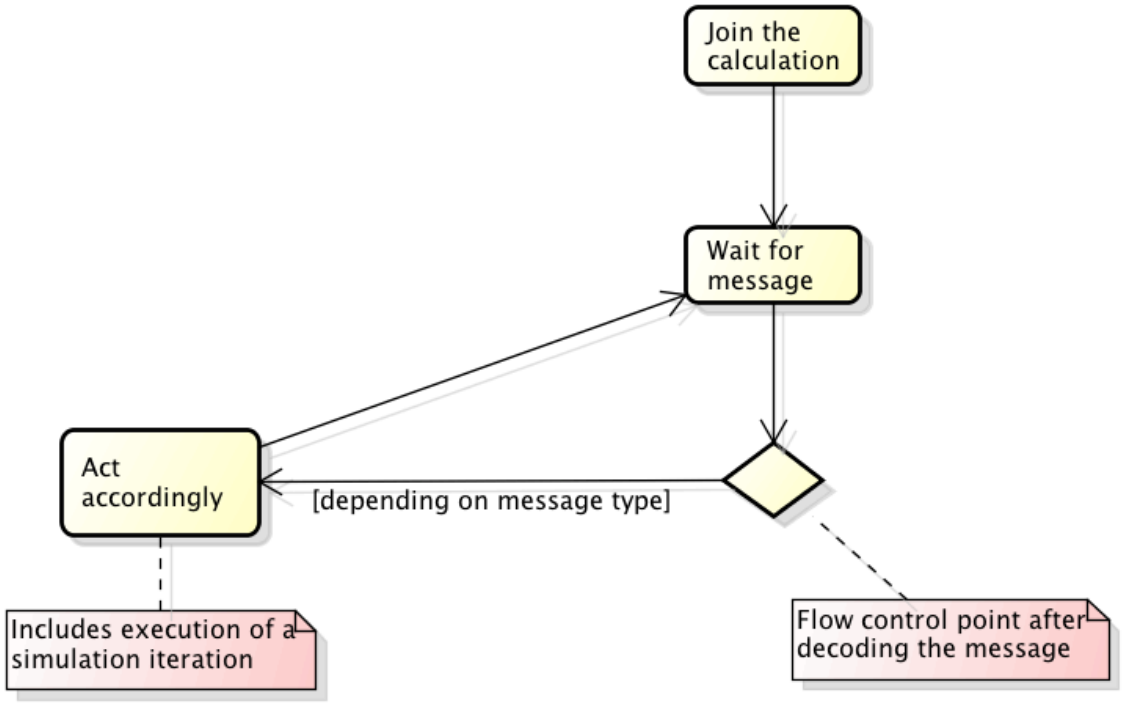

Ilustración 13 - Diagrama básico de funcionamiento de un nodo

En todos los casos la programación se ha hecho con un único hilo, por lo que perdemos la oportunidad de tener un rendimiento mejor.

- Utilización de hilos en los nodos

3.2.2.1. Maestro

Este nodo es el que empieza la computación. Sus responsabilidades son muy básicas, y se resumen fácilmente en el siguiente diagrama.

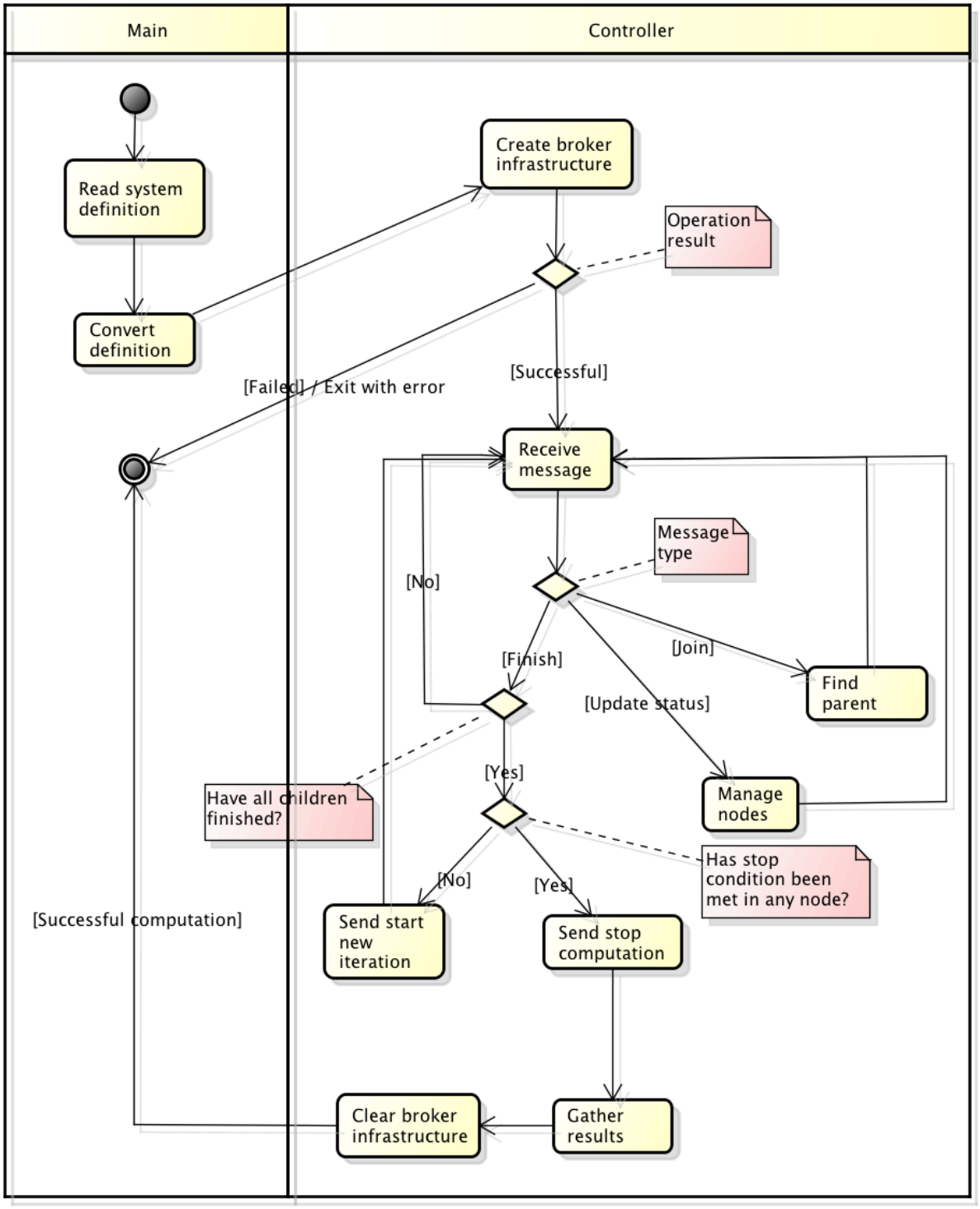

Ilustración 14 - Diagrama de actividad de un nodo maestro

Vemos como este tipo de nodo sigue el esquema general presentado antes. Los mensajes que recibe y procesa este tipo de nodo son más de los que se muestran en la imagen. Aquí sólo mostramos los mensaje principales para el sistema, que van a apoyar las operaciones de:

- Unión de nuevos nodos a la simulación

- Coordinación para realizar cambio de iteración
- Mantenimiento de nodos hijo y su información
- Búsqueda de un padre para un nodo que se acaba de unir al cálculo

El resto de mensajes que recibe este nodo son para llevar a cabo alguna de las tareas antes mencionadas.

Como se comentó en la sección anterior, la implementación se lleva a cabo con un único hilo. Ello evita resolver el problema del acceso a recursos compartidos. Dichos recursos son en este caso:

- Atributos del *NodeManager*
- Dominio (membranas de las que el nodo es responsable)

Para mejorar esta faceta deberíamos fijar los siguientes objetivos.

- Separar a otra clase el procesado de mensajes con un *AMQPChannel* para sí misma.
- Ejecutar varias instancias de dicha clase dentro del controlador, una en un hilo diferente.
- Proteger el acceso concurrente a los recursos compartidos mencionados más arriba.

En el diagrama anterior, hay un paso en el controlador en el que se recogen los resultados. Actualmente se realiza de forma normal: creando un objeto String en memoria a su llegada al nodo. Esto resta toda ventaja del sistema de poder hacer que todo nodo sea inconscientes del resto. Así, proponemos una mejora, que también aplica a los nodos proxy.

- La recepción de datos se hará por un stream que se redirigirá a un fichero, evitando la creación en memoria de un String.

3.2.2.2. Esclavo

Los nodos de este tipo son las hojas del árbol que forma el conjunto de todos los nodos de un sistema. Esto es debido a que son nodos que no pueden tener hijos.

Al tener esta característica, no es necesario para ellos tener un bróker de mensajería instalado.

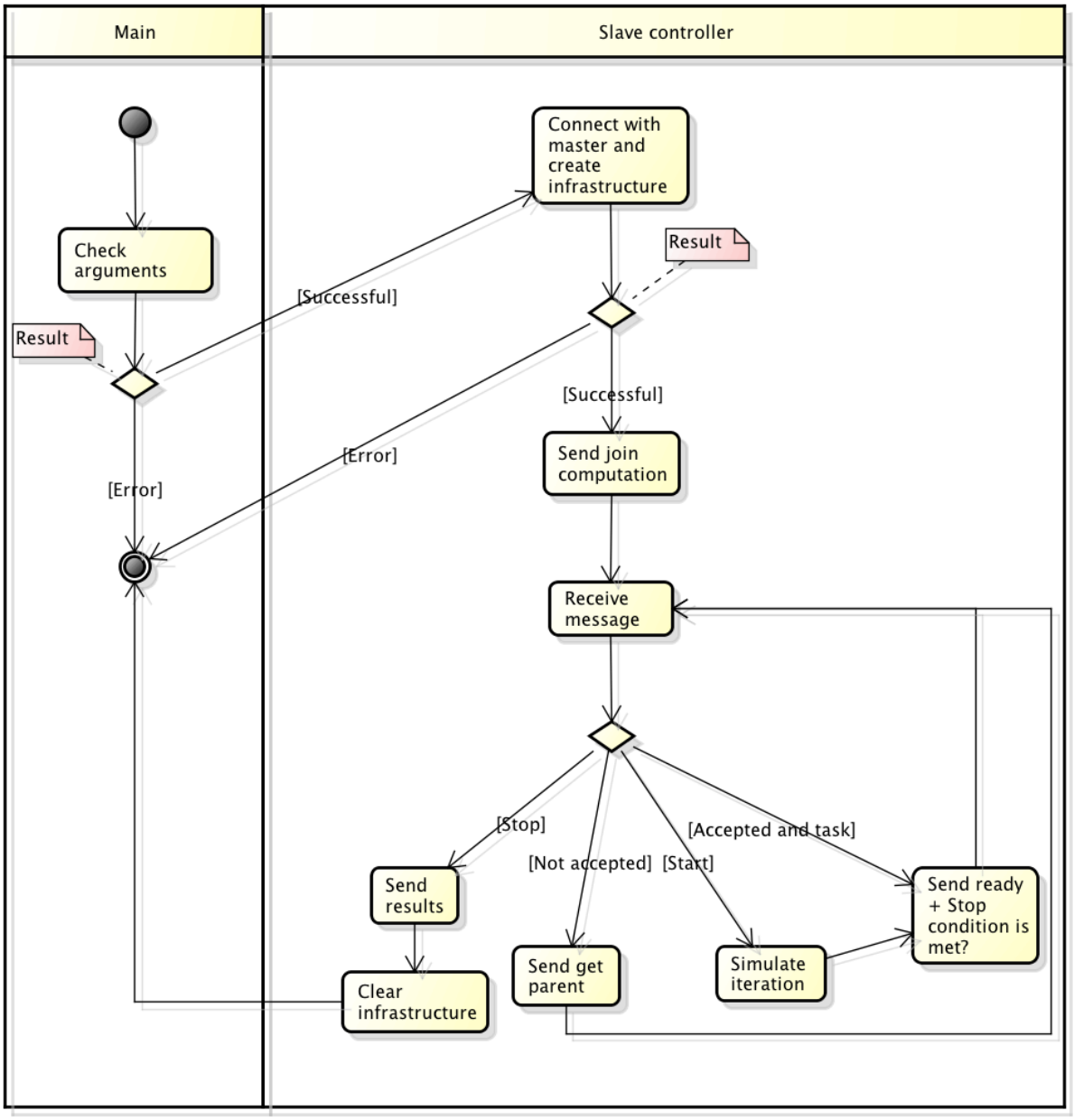

Ilustración 15 - Diagrama de actividad de un nodo esclavo

Los nodos esclavo tienen una lógica más sencilla que la de los nodos que tienen hijos, ya que no necesitan coordinarse con otros nodos, sino que

simplemente se dedican a recibir órdenes y ejecutar acciones. Además del único entorno que son conscientes es de su nodo padre, que por el momento está restringido a uno (esto aplica también a los nodos proxy, los cuales sólo pueden tener un padre).

- Permitir que un nodo (proxy o esclavo) tenga más de un padre

3.2.2.3. Proxy

Los nodos proxy comparten características con los nodos maestro y esclavo. Así, son hijos de otros nodos (como los esclavo), y pueden ser padres de otros nodos (como el nodo maestro). Esto se muestra en el siguiente diagrama, el cual está bastante resumido para evitar repetir conceptos.

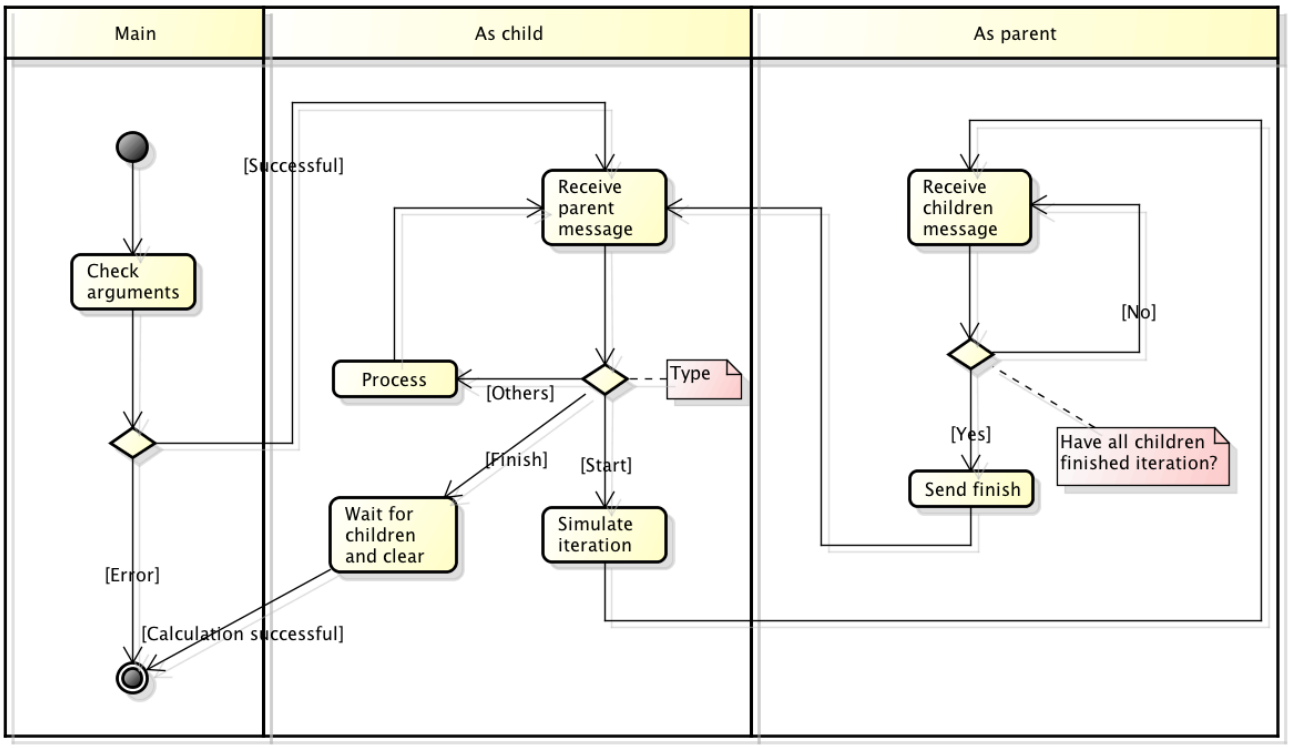

Ilustración 16 - Diagrama de actividad resumido de un nodo proxy

En este diagrama resumido podemos ver cómo se combinan las dos facetas de un nodo proxy: hijo de un nodo y padre de varios.

En este caso la secuencialidad del algoritmo no lo hace ni mucho menos óptimo, por lo que proponemos una mejora en este sentido.

- Refactorizar y tratar de poner un lugar común el código que comparten los nodos proxy con los nodos maestro y esclavo.

- Ejecutar en dos hilos (y dentro de cada uno de ellos otros tantos hilos) diferentes las funciones de nodo padre y las de nodo hijo de un proxy.

3.2.3. Manejo del bróker

En este apartado sólo destacar que hemos creado un envoltorio alrededor de las llamadas a la API del bróker, para que de esta manera nuestro simulador sea más independiente de la tecnología subyacente.

De esta manera un cambio en la API o incluso si quisiésemos cambiar el bróker (y por tanto las interfaces cambiarían, pero no los conceptos manejados) por otro diferente que cumpla con la especificación AMQP (http://amqp.org/) conllevaría menos modificaciones en nuestro código.

3.2.4. Comunicaciones en distribuido

Para la simulación de sistemas en un entorno distribuido, las operaciones entre membranas han de ser las mismas. Esto se hace mediante paso de mensajes entre nodos. Dichos mensajes pertenecen a un protocolo que hemos diseñado y que se explica en la siguiente sección.

Los mensajes se manejan en objetos dentro del simulador, y para enviarlos hay que empaquetarlos en origen y desempaquetarlos en destino. El tipo de paquete y el proceso de empaquetado y desempaquetado pueden ser un factor importante para el rendimiento, por lo que este es otro de los puntos que son fácilmente extensibles.

- Terminar de implementar el empaquetado de mensajes con **J**ava**S**cript **O**bject **N**otation (http://www.json.org/).

El sistema tiene que asegurar un cierto orden en el paso de mensajes que conciernen a la comunicación e intercambio de elementos entre membranas.

De esta forma, al tener la red de nodos una estructura de árbol, simplemente hemos de hacer que un nodo padre no expulse membranas ni símbolos a su membrana padre hasta haber recibido las membranas y símbolos de sus membranas hijas (lo mismo que había que hacer en el simulador local).

También aseguramos así que todos los nodos ejecutan la misma iteración a la vez, al no enviar un nodo el mensaje de *Ready for calculation* hasta que no ha recibido el mismo de todos sus hijos. La orden de comenzar el cálculo de una nueva iteración baja al final desde el nodo raíz hasta las hojas.

Cabe mencionar aquí cómo se comunica un nodo con su padre. Lo hace a través de lo que llamamos membranas remotas. Una membrana remota no es sino una membrana sobre la que no se ejecutan reglas, ya que es como un puntero a una membrana situada en otro nodo (nodo padre). De esta forma sirve de contenedor para los símbolos o membranas que se pueden expulsar a la membrana padre en cada iteración. Es importante destacar que una membrana remota siempre será una de las membranas más externas en un nodo. El siguiente diagrama ilustra esta situación.

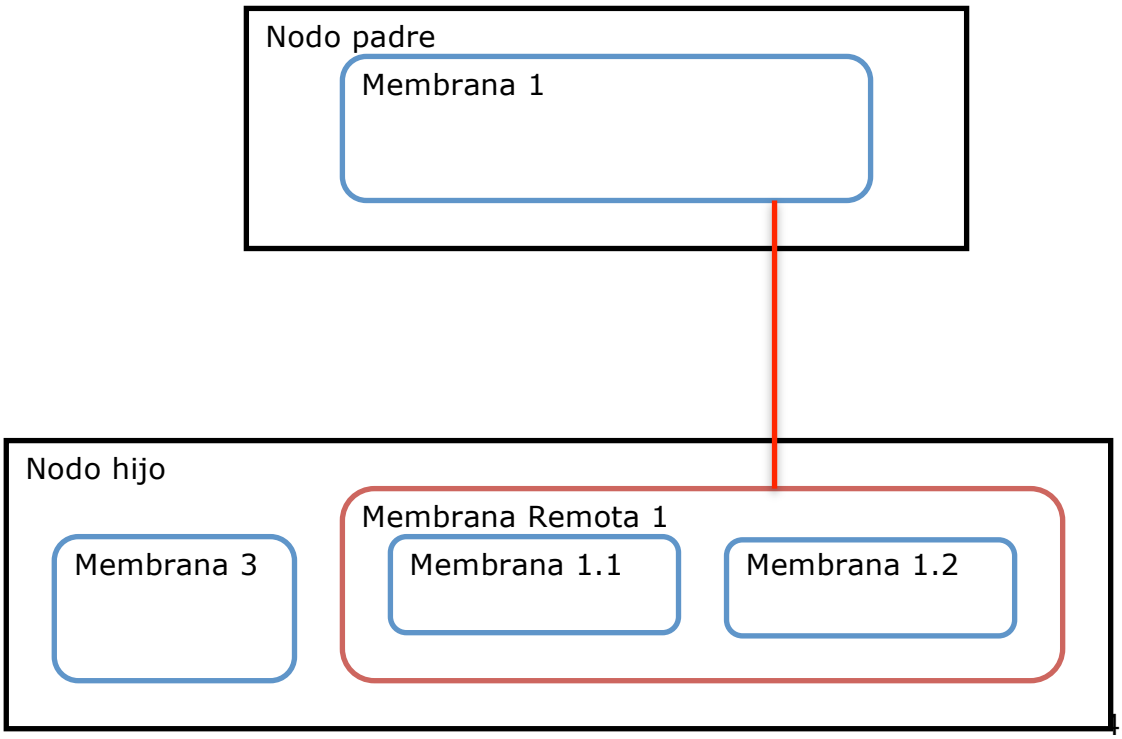

Ilustración 17 - Membranas remotas

3.2.4.1. Protocolo

La comunicación y coordinación entre nodos están regidos por un protocolo diseñado por nosotros. Este protocolo es fácilmente extensible, y para crear un nuevo tipo de mensaje sólo se requiere de:

- Definición del mensaje y su contenido
- Manejo y creación del mismo en los controladores de nodo
- Empaquetado y desempaquetado para enviarlo a través de la red

Los tipos de mensaje del protocolo se pueden agrupar por operaciones que se consiguen hacer usándolos. Dichas operaciones requieren de cierta sincronía entre los nodos a través del paso asíncrono de estos mensajes.

Las operaciones más importantes son:

- *Hand shake* → adición de un nodo a la computación
- *Partition* → envío de trozos del problema a resolver a otros nodos (es también parte del *hand shake*)
- *Report status* → un nodo envía información a su padre, que éste utilizará para colocar nuevo nodos
- *Operativa* → mensajes de coordinación para ejecución de iteraciones de cálculo

3.2.5. Operaciones más relevantes

En esta sección presentamos las operaciones más relevantes en el ámbito de la ejecución distribuida de problemas. También mostraremos cómo se coordinan los nodos para llevar a cabo dichas operaciones.

3.2.5.1. Adición de un nuevo nodo

La secuencia de mensajes para añadir un nuevo hijo a la computación se describe en los siguientes diagramas:

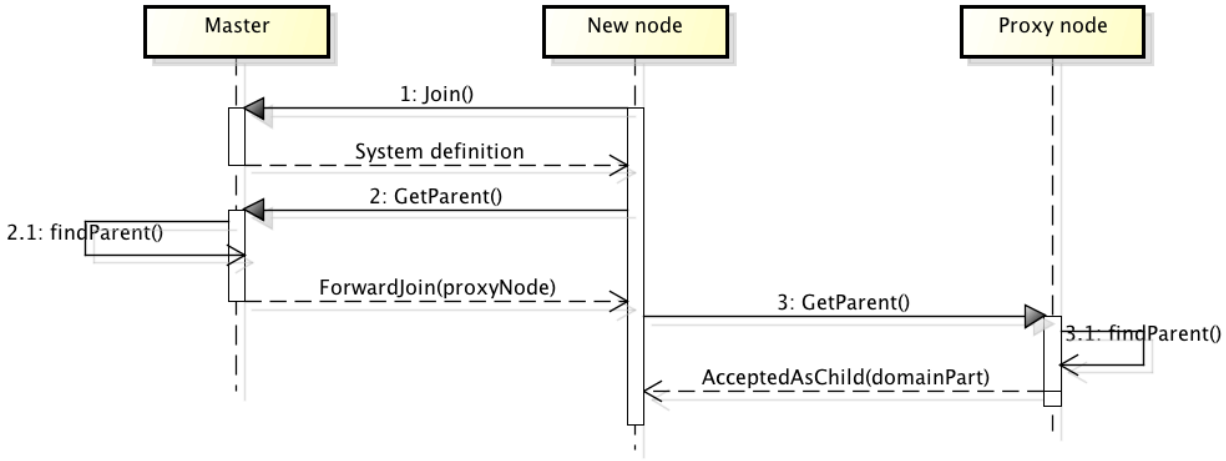

Ilustración 18 – Nodo no acogido por el nodo maestro y enviado a un nodo proxy

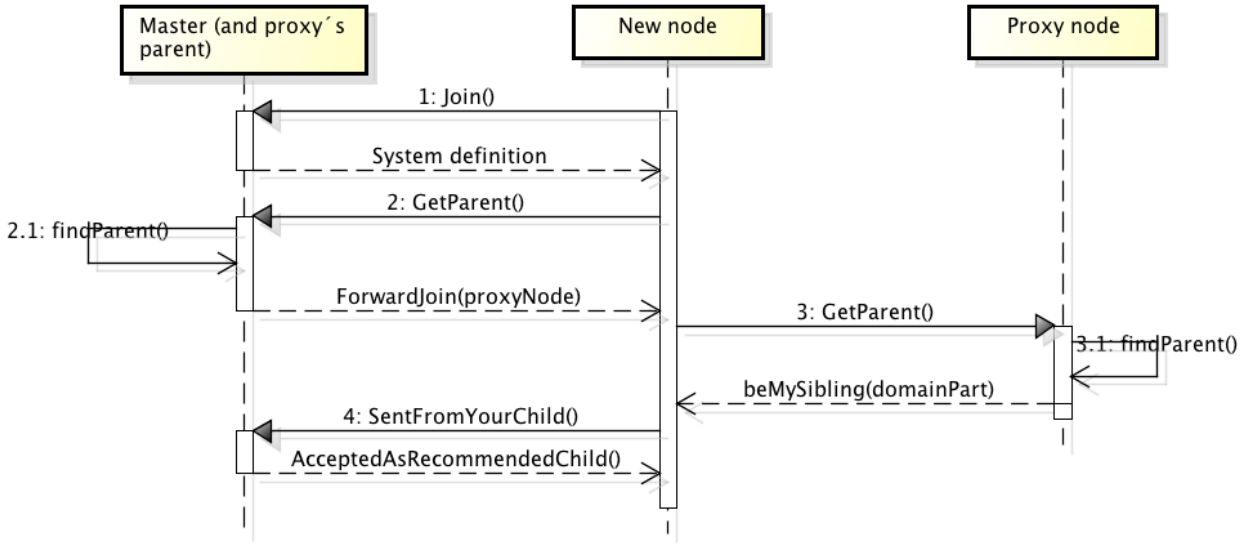

Ilustración 19 – Nodo enviado al nodo padre para expandir horizontalmente el árbol y aprovechar la potencia del broker

La estrategia en la colocación de nuevos nodos dentro del árbol tiene como objetivos:

- aprovechar la potencia del bróker de mensajería
- que la coordinación de las operaciones se haga con el menor número posible de mensajes

El **primero** de los objetivos persigue a su vez dejar las tareas más pesadas y no interesantes para el dominio de los sistemas P (estos es, las comunicaciones) en manos de una tecnología usada en entornos de

producción en el mundo empresarial. Ello se consigue teniendo bajo cada nodo proxy el mayor número posible de hijos. Nuestra operativa en este sentido depende en gran medida de la partición de dominio, que se explicará en profundidad en el siguiente capítulo.

- Refactorizar el particionador de dominio y el manejador de nodos para que estén más interrelacionados, ya que esa parece ser su naturaleza.

El **segundo** objetivo se alcanza teniendo un árbol lo menos profundo posible por lo que se consigue a través del primero de los objetivos.

En cualquier caso es fácil modificar o extender el funcionamiento de la colocación de nodos, ya que como mencionamos al principio se han intentado abstraer interfaces en las partes de la aplicación que son susceptibles de ser modificadas para probar varias estrategias, y éste es uno de esos casos.

La operativa de colocación de nodos sólo se ejecuta cuando un nodo se une a la simulación, pero sería deseable utilizarla para balacear el árbol a intervalos regulares (cada *n* iteraciones) o cuando un nodo se una a la computación (esta mejora ya se ha propuesto en el apartado de funcionalidad).

3.2.5.2. Partición de dominio

Esta operación consiste en trocear el problema, es decir, las membranas a simular. En nuestro caso se hace en los nodos de forma local. El resultado se envía, eso sí, a otro nodo receptor de la partición, sobre la que ejecutará la simulación.

La estrategia que seguimos es: si un nodo puede particionar el dominio que maneja para que le nodo receptor sea su hermano, eso hará. De esta manera conseguimos expandir el árbol horizontalmente y evitar que se haga más profundo. Así, como ya comentamos con anterioridad, aprovechamos la capacidad del bróker de mensajería.

En caso de no poderse particionar de esa forma, entonces se dará un trozo del dominio al nuevo nodo para que se acabe convirtiendo en su hijo.

Esto era en cuanto a la relación de las membranas del nodo respecto a las de la partición resultante. Respecto del número de membranas que un nodo da a otro al particionar el dominio:

- Si el nuevo nodo será hermano, éste recibirá la mitad de los subárboles
- Si el nuevo nodo será hijo, recibirá todo el subárbol con raíz en la membrana que se quedará el nodo que hace la partición

4. Resultados

Como prueba de funcionamiento correcto del sistema hemos ejecutado instancias de diferente tamaño de un problema bien conocido problema de la mochila, que ha sido modelado en [6]. El objetivo de estas pruebas era detectar y depurar errores y comprobar la correcta consecución de los objetivos sobre todo en lo relacionado al acceso a ordenadores a través de internet y el mantenimiento del rendimiento cuando se escala el tamaño del problema.

Tras a ejecución de las pruebas vemos que hemos cumplido los objetivos propuestos al principio del proyecto: ejecución distribuida del simulador, y no dependencia del tamaño del problema (exceptuando en la recepción de resultados en los nodos, quedando esa mejora como una prioridad para el futuro, y sabiendo que no es una tarea compleja ni que requiera de mucho tiempo).

En cuanto a tiempo de ejecución, la resolución de un problema tarda aproximadamente igual, o un poco menos, en un entorno distribuido (3 ordenadores) a pesar de tener que compartir la CPU con el bróker de mensajería. La configuración distribuida que hemos utilizado ha sido la siguiente:

- Ordenador 1

    - Nodo maestro
    - Nodo proxy 1
- Ordenador 2
    - Nodo proxy 2
    - Nodo esclavo 1
- Ordenador 3
    - Nodo esclavo 2

Los ordenadores eran todos de características HW dispares, e incluso sistemas operativos diferentes.

Es importante resaltar que la ejecución en local del mismo modelo se hizo en el más potente de los ordenadores, y como mencionamos antes el tiempo de ejecución fue incluso un poco menor en la ejecución distribuida.

5. Líneas futuras

En un futuro nos gustaría seguir trabajando en el simulador para que:

- Pueda soportar todas las operaciones al simular en distribuido
- Tenga robustez
- Sea capaz de recuperarse de fallos en nodos

Estos objetivos han de cumplirse siguiendo los siguientes pasos:

1. Realizar las mejoras enmarcadas en rojo a lo largo de esta memoria
2. Realizar las mejoras enmarcadas en naranja a lo largo de esta memoria
3. Crear la implementación de las operaciones que falten para los modelos que queramos simular
4. Realizar las mejoras enmarcadas en verde que se deseen

Respecto al sistema que nos gustaría modelar entrarían los que muestren algún tipo de comportamiento emergente (como pueden ser huídas de inversores en bolsa), o algún tema relacionado con el medio ambiente.

En estos momentos estamos buscando algún grupo, institución o incluso empresa que esté interesada en utilizar el simulador para simular algún sistema real (aunque no sean los anteriormente descritos).

Una buena idea para hacer un uso masivamente paralelo del sistema sería ejecutarlo en la nube, y ese sería otro paso hacia el futuro del simulador. Y esto combinado con un sistema real (que muy seguramente tendría grandes dimensiones) nos daría una idea muy buena del rendimiento del simulador y por tanto si merece la pena seguir trabajando en esta línea.

6. Bibliografía

1. Anderson, D. J. (2010). KANBAN. Successful Evolutionary Change for Your Technology Business. Washington, USA: Blue Hole Press.
2. Cecilia, J. M., Guerrero, G. D., García, J. M., Martínez del Amor, M. A., Pérez Hurtado, I., & Pérez Jiménez, M. J. (2009). Simulation of P Systems with Active Membranes on CUDA. *International Workshop on High Performance Computational Systems Biology.*
3. De Frutos Velasco, J. A. (2012). Análisis estático de sistemas de membranas para la determinación de reglas activas.
4. Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters.
5. Diez Dolinski, L., Núñez Hervás, R., Cruz Echeandía, M., & Ortega de la Puente, A. (2011). Distributed Simulation of P Systems by Means of Map-Reduce: First Steps with Hadoop and P-Lingua. *Lecture Notes in Computer Science, 6691*, ss. 457-464.
6. Jiménez, M. J., & Riscos Núñez, A. (2004). A linear-time solution to the knapsack problem using active membranes. *Lecture Notes in Computer Science, 2933*, 250-268.
7. Navarrete, C. B., & Anguiano, E. (2009). From static domains to graph decomposition for heterogeneous cluster programming. *High Performance Computing on Vector Systems*.
8. Paun, G. (2000). Computing with membranes. *Journal of Computer and System Sciences*, ss. 108-143.
9. Peña Camacho, M. A. (u.d.). Algoritmos de distribución de cargas de proceso en computación con membranas.
10. Poppendieck, M., & Poppendieck, T. (2006). *Implementing Lean Software Development: From Concept to Cash.* (A.-W. Professional, Red.)

11. Riscos Núñez, A. (2004). Programación celular: resolución eficiente de problemas numéricos NP-completos. Sevilla.
12. Syropoulos, Apostolos and Mamatas, Eleftherios G. and Allilomes, Peter C. and Sotiriades, Konstantinos T. (2004). A Distributed Simulation of Transition P Systems. *Lecture Notes in Computer Science*, vol. 2933, pages 133-145. Springer Berlin
13. Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, University of Tennessee, UT-CS-94-230 (1994)
14. L. Fernández, V. J. Martínez, and L. F. Mingo (2005). A hardware circuit for selecting active rules in transition p systems. *In Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2005)*.
15. Garcia-Quismondo Fernández, M., Gutiérrez Escudero, R.M., Pérez Hurtado de Mendoza, I., Perez Jimenez, M.J., Riscos Núñez, A.: An Overview of P-Lingua 2.0. *Lecture Notes in Computer Science.* Nm. 5957. 2010. Pag. 264-288
16. Gh. P_aun, G. Rozenberg, and A. Salomaa. (1998) DNA Computing. New Computing Paradigms. Berlin, Springer.