Proyecto final - "Reparación de programas distribuida"

Stryker distribuido

Francis Iván Gilly
legajo 54053
fgilly@itba.edu.ar

Fernando Bejarano González
legajo 52043
fbejaran@itba.edu.ar

Marcelo F. Frias (Director) mfrias@itba.edu.ar

Luciano Zemin (co-Director) lzemin@itba.edu.ar

17 de noviembre de 2017

Resumen

La herramienta "Stryker" implementa una técnica para reparar programas equipados con contratos. Esta técnica combina análisis dinámico (en tiempo de ejecución) con análisis estático para verificar las reparaciones candidatas y emplea un mecanismo para detectar y podar candidatos no viables aprovechando las especificaciones del programa.

En este artículo se presenta "Stryker distribuido", una herramienta prototipo que implementa una mejora a la técnica de Stryker para permitir su escalamiento a múltiples computadoras. En esta nueva técnica se desarrolló un mecanismo de poda distribuida.

La técnica fue evaluada para comparar su funcionamiento en múltiples computadoras utilizando un benchmark que pertenece a Stryker y que consiste en un conjunto de clases de Java con fallas. Los experimentos muestran que en general el tiempo de ejecución disminuye a medida que se aumenta la cantidad de computadoras que se utilizan. También se demostró que a mayores cantidades de fallas en el programa, se obtienen mayores reducciones en cuanto al tiempo.

Índice

1. Introducción **3**

2. Requerimientos del proyecto. **3**
2.1. Funcionales . . . 3
2.2. No funcionales . . . 3

3. Algoritmos **3**
3.1. Conceptos básico de Stryker . . . 3
3.2. Algoritmo de Stryker original . . . 4
3.3. Stryker distribuido . . . 4

4. Arquitectura **9**
4.1. Arquitectura de aplicación . . . 9
4.2. Controllers . . . 9
4.2.1. Controllers esclavos . . . 12
4.2.2. Controllers maestros . . . 12
4.3. Mensajes . . . 13
4.4. Comunicación Master-Slave . . . 13

5. Funcionamiento **14**
5.1. Encendido . . . 14
5.2. Solución . . . 14
5.3. Apagado . . . 14
5.4. Reinicio de esclavos/cancelación de tareas . . . 14

6. Dificultades encontradas **14**

7. Evaluación **15**
7.1. Especificación de casos de prueba . . . 15
7.2. Resultados experimentales . . . 15
7.2.1. Peores casos de la versión distribuida . . . 15
7.2.2. Mejores casos de la versión distribuida . . . 16
7.2.3. Comparación peores y mejores casos . . . 18
7.2.4. Tiempos promedios . . . 18
7.2.5. Distribución de speed ups . . . 19
7.3. Optimización . . . 20
7.3.1. Análisis de cuello de botella en la creación de particiones . . . 20
7.3.2. Análisis de cuello de botella en el envío de particiones . . . 24
7.3.3. Solución del cuello de botella de la creación de particiones . . . 26

8. Conclusión **29**

9. Trabajos futuros **30**
9.1. Mejora en la división de tareas . . . 30
9.2. Apagado automático . . . 30
9.3. Tolerancia a falla de esclavos . . . 30

10. Referencias **31**

1. Introducción

Stryker es una herramienta para realizar la reparación automática de programas que incluye una técnica novedosa para podar el espacio de candidatos que deben ser considerados como reparaciones. Esta herramienta utiliza la librería MuJava[1] para obtener los programas candidatos a partir de la mutación del código original que presentaba errores, y emplea la herramienta Taco[2] para la detección de fallas y la validación de las soluciones mediante la verificación del contrato JML que posee la clase a reparar. También utiliza RAC[3] para descartar rápidamente programas candidatos que son incorrectos en tiempo de ejecución. Con su técnica de poda, Stryker logra una gran efectividad en la reparación de programas. Aún así, sigue requiriendo bastante tiempo para encontrar la solución para código más complejo y con varios errores.

MuJava es una librería para realizar modificaciones a las líneas del código de un programa escrito en Java. Utiliza un amplio conjunto de operadores de mutación intra-sentencia que producen variantes sintácticas de una línea de código. Entre estos operadores se destaca el mutador PRV que dada una expresión de Java donde se accede a variables anidadas del tipo f1.f2...fn, genera todas las posibles combinaciones con los nombres de forma de obtener expresiones bien tipadas.

RAC (Runtime Assertion Checker) es una librería que forma parte del conjunto de herramientas JML y que permite ejecutar con ciertos inputs a un método de una clase junto con su contrato JML.

Taco (Translation of Annotated COde) es una herramienta que implementa una técnica general y completamente automatizada de análisis basada en SAT sobre código secuencialmente anotado que involucra a complejas estructuras de datos vinculadas. Esta herramienta permite realizar la localización de fallas sobre código anotado con contratos JML (Java Modeling Language). Para realizar esto, utiliza el código fuente de un método en una clase de Java junto con el correspondiente contrato JML y lo reduce a un problema de satisfacibilidad booleana.

Si bien los análisis con TACO son efectivos para determinar si un programa tiene fallas, los mismos son costosos en tiempo. Es por eso que Stryker utiliza la herramienta RAC para eliminar rápidamente a los programas candidatos como una primera instancia de filtrado. Para esto, además del programa candidato junto con el correspondiente contrato JML, también se utilizan los inputs que hicieron fallar a los programas candidatos en análisis hechos previamente con TACO.

En la sección 3 se presenta un algoritmo que distribuye el proceso realizado por Stryker, luego en la sección 4 se detallará la implementación realizada en Java, y en la sección 7 se comentarán los resultados de una evaluación de la herramienta utilizando un conjunto de pruebas de clases de Java con fallas.

2. Requerimientos del proyecto.

2.1. Funcionales

En este proyecto se requiere distribuir el proceso de reparación de programas de forma que la distribución sea compatible con la técnica de poda. En este sentido, se requiere desarrollar el algoritmo de distribución e implementar el mismo en una versión distribuida de Stryker.

Adicionalmente, se requiere que la mayoría de los procesos de Taco se ejecuten en los servidores esclavo ya que es el paso más costoso del algoritmo de Stryker.

2.2. No funcionales

- **Escalabilidad**
  El software desarrollado debe funcionar en un clúster de computadoras y escalar a todo un laboratorio de informática de la universidad.
- **Debe funcionar en el sistema operativo Linux.**
  Este sistema operativo es el que poseen las computadoras de los laboratorios de informática.
- **Debe correr sobre máquina virtual Java.**
  Las modificaciones se deben realizar sobre un proyecto desarrollado previamente en Java.
- **Eficiencia.**
  Se necesita aprovechar los recursos de los múltiples servidores esclavos al máximo. En este sentido, siempre que sea posible se deben distribuir tareas para su procesamiento en paralelo.
- **Correctitud.**
  Se debe realizar un testeo abundante pero la corrección es relativa a las herramientas que se utilizan, las cuales por su complejidad pueden contener bugs.
- **Soportabilidad.**
  Es necesario contar con suficiente información sobre la ejecución del software para poder identificar y resolver fallas, y para comprender su funcionamiento.

3. Algoritmos

3.1. Conceptos básico de Stryker

El concepto fundamental con el que trabaja Stryker es el de “mutante”. Un “mutante” es el código de la clase a testear con uno o más cambios realizados para intentar reparar los errores que tiene. Se lo identifica con un vector cuyos

elementos son los índices de los operadores de mutación aplicados en cada sentencia. Dicho vector tiene el siguiente aspecto:

$(mutación_1, mutación_2, \ldots, mutación_N)$

De esta manera si se lee el vector de izquierda a derecha el primer elemento del mismo ($mutación_1$) corresponde con la sentencia más cercana al final del método a reparar que puede tener un bug en la clase original, y el último elemento de dicho vector ($mutación_N$) es la sentencia más cercana al comienzo del método que puede tener un bug.

Por otra parte, un "padre" es un mutante con la particularidad de que se le pueden aplicar nuevas modificaciones en sentencias que ya fueron modificadas previamente o que están por ser modificadas por primera vez.

Además, Stryker utiliza como algoritmo de poda de mutantes al proceso "getFeedback" que se encuentra detallado en [4]. Este proceso permite obtener la cantidad de sentencias mutables cuyas mutaciones pueden saltearse. Para calcular esto, se utiliza internamente el concepto de "K-variabilización", que consiste en cambiar los términos por variables nuevas en las sentencias desde el final del método hasta la sentencia número K contabilizada desde el final. Estas variables nuevas quedan libres para que el SAT-solver que utiliza Stryker les asigne valores con el fin de evaluar si en dicha sentencia existe algún valor que permita arreglar los bugs contenidos en dicho método.

En caso de que no se pueda solucionar los bugs hasta la sentencia K, se intenta en la sentencia superior. Este proceso se detiene cuando se llega a una sentencia que posibilita arreglar los bugs.

Con este resultado, se puede asegurar que la solución no se encuentra mutando solamente hasta la sentencia $K-1$ sino que es necesario mutar hasta la sentencia K como mínimo. De esta manera se pueden podar los mutantes que solo presentan mutaciones hasta la sentencia $K-1$.

3.2. Algoritmo de Stryker original

El funcionamiento del algoritmo de Stryker original se encuentra graficado en la figura 1. Primero se determina si el método de una clase tiene fallas. Si este es el caso, se procede a localizar las sentencias que pueden presentar fallas.

Una vez que se identificaron las sentencias sospechadas de tener fallas, se generan mutantes utilizando la herramienta MuJava y se guardan en una cola. Luego a cada mutante se lo evalúa para ver si es solución.

Un mutante es solución si primero pasa los contraejemplos (análisis con herramienta RAC), luego pasa un análisis estático de código con la herramienta Taco, y por último pasa un análisis con todos los contraejemplos obtenidos. Por otra parte, si no es solución se calcula el valor de "feedback" y se guarda en la cola el siguiente mutante relevante que se genera a partir de este mutante aplicando las mutaciones a izquierda y a derecha que corresponden.

El ciclo principal de este algoritmo funciona levantando mutantes de una cola y aplicándole a cada uno este análisis hasta encontrar una solución o agotar el espacio de búsqueda.

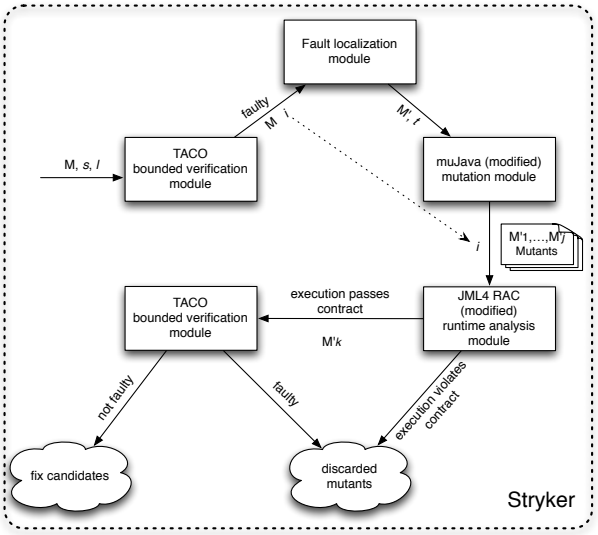


Figura 1: Descripción esquemática de Stryker [4].

3.3. Stryker distribuido

En el algoritmo 1 se encuentra la idea principal de la versión distribuida de Stryker. Este algoritmo se ejecuta en el servidor maestro. Comienza tomando un mutante de la cola de métodos con mutaciones pendientes. Después, se obtiene el último valor de feedback almacenado para dicho método (línea 5). Dicho valor es calculado en los esclavos utilizando el método "getFeedback()"; esto es para delegar este proceso costoso en los esclavos y reducir la carga de trabajo del servidor maestro.

Más adelante, calcula un lote de mutantes a distribuir entre los servidores esclavos. Para armar dicho lote se utiliza el valor de feedback obtenido previamente y en la posición que indica este valor se asigna un índice de mutación en el arreglo de mutaciones que queda fijo para cada esclavo (línea 9). Los valores del arreglo de mutaciones que están antes de la sentencia de feedback quedan libres para que se prueben las combinaciones en el esclavo.

Por ejemplo para un método a reparar que tenga 5 sentencias con bugs y para el cual el valor de feedback indica la cuarta posición en el vector de mutaciones, un lote de particiones a enviar es el siguiente:

$$\begin{array}{l} partición_1 : (A_1 \quad A_2 \quad A_3 \quad \mathbf{1} \quad 0) \\ partición_2 : (B_1 \quad B_2 \quad B_3 \quad \mathbf{2} \quad 0) \\ \qquad\qquad\qquad\qquad\qquad\quad \vdots \\ partición_N : (Z_1 \quad Z_2 \quad Z_3 \quad \mathbf{N} \quad 0) \end{array} \qquad \begin{array}{r} 0 \leq A_i, B_i, \ldots, Z_i \leq \text{Máximo índice de mutación para sentencia i} \\ N: \text{ Máximo índice de mutación para posición 4} \end{array} \tag{1}$$

En este ejemplo, cada vector representa a una partición y en la quinta posición de cada vector quedó fijo el valor cero que se corresponde con la no aplicación de una mutación en dicha sentencia. Además, para cada esclavo la cuarta posición de los arreglos queda fija en un índice. Por último, cada arreglo es asignado posteriormente a un servidor esclavo distinto en el cual se van a variar las posiciones 1, 2 y 3 con los índices de las mutaciones posibles para cada sentencia.

El algoritmo 2 se ejecuta en el servidor maestro, realizando dos tareas en paralelo para maximizar el uso de los servidores esclavos: envío de particiones a los servidores esclavos y el procesamiento de las respuestas provenientes de los mismos.

Para realizar la distribución de las particiones (líneas 3 - 5), primero se obtiene una tarea (línea 3), luego se busca un servidor esclavo libre (línea 4) y por último se envía esa tarea a dicho esclavo (línea 5). En la segunda parte del algoritmo (líneas 9 - 18), se analizan las respuestas provenientes de los esclavos. En dicha parte, se itera en principio hasta recibir una respuesta por cada partición enviada del lote. Cuando se recibe una respuesta (línea 10) se procede a guardar los contraejemplos generados en el esclavo (línea 12); este paso es importante ya que se necesitan estos contraejemplos para realizar una última pasada del análisis RAC sobre el mutante candidato (algoritmo 3 línea 14) y también son útiles para enviar a los esclavos para descartar mutantes rápidamente en el análisis RAC (algoritmo 3, línea 8)

Si esta respuesta es solución, se finaliza la ejecución. Si por otra parte se recibe un valor de "getFeedback" mayor al utilizado para generar la partición, esto quiere decir que se pueden saltear las particiones de este lote ya que se necesita mutar en una sentencia superior entonces se cancelan las particiones de este lote (algoritmo 2, líneas 16-17)

En caso de que se hayan recibido todas las respuestas del lote y no hayan ocurrido alguno de los casos anteriores, es decir que se agotó el espacio de búsqueda sin encontrar un valor de feedback superior, entonces se indica que es necesario mutar una sentencia superior (línea 19).

El algoritmo 3 se aplica luego de recibir la tarea en cada esclavo para trabajar con las posibles mutaciones de la partición que le fue asignada. Dentro de cada partición, se itera por los mutantes relevantes mientras no se encuentra solución. Es por esto que después de evaluar cada mutante, se computa getFeedback para podar mutaciones. Como resultado del procesamiento de cada partición, pueden ocurrir tres casos:

1. Se encuentra una solución (línea 16). En este caso tiene que ocurrir que el mutante pase el primer análisis de RAC con los contraejemplos que tenía el esclavo (recibidos desde master con la partición junto con los generados en el esclavo), el análisis de Taco y un análisis de RAC en el servidor maestro hecho con todos los contraejemplos recolectados en el servidor maestro hasta el momento.
2. Se termina de evaluar la partición sin encontrar solución. Esto puede ocurrir porque el cálculo del valor de feedback dio en la posición del arreglo de mutaciones en la que está limitada la partición (línea 28). En este último caso es necesario podar hasta el final de la partición. También puede ocurrir que no se encuentre solución porque se evaluaron a todos los mutantes relevantes (línea 29).
3. Se obtuvo un valor de feedback superior a la posición del arreglo a la que está limitado. En este caso debe avisar al servidor maestro para que pueda podar el lote de particiones completo (línea 28).

Algorithm 1: [Master server]Pruning along mutations traversal

Input: Q, cola de métodos con mutaciones pendientes;
Input: S, set de inputs que satisfacen precondiciones de M
Input: m, cantidad total de sentencias mutables
Result: M' (primer arreglo exitoso encontrado o "not fixable" si no es posible arreglarlo)

```
1  Method M= Q.pop();
2  k_1, ..., k_m =M.getNumberOfMutationsPerMutableStatement();

3  boolean fixFound=false;
4  ArrayIterator iterator=new ArrayIterator( k_1, ..., k_m ) ;          // Iterador de índices de mutaciones a aplicar

5  int k=M.feedBack ;      // Cantidad actual de sentencias que se pueden saltear(desde el final del método). Va de 0 a
   (m-1).
6  while ! fixFound && k < m && iterator.hasNext() do
       /* Podado de mutaciones con k-variabilización                                                          */
7      iterator.setToZeroAllPositionsInRange(0, k-1);
8      iterator.advanceOneStartingAtIndex(k);

       /* Conjunto de iteradores de particiones para enviar.                                                  */
       /* Mismo iterador de antes, pero que llega hasta sentencia k. No modifica sentencias más arriba.Cada iterador tiene
          en la línea k un valor fijo, y va variando (empezando por cero) las sentencia de abajo.             */
9      List < LimitedArrayIterator > partitionIterators= iterator.kLimit(k);

       /* Esta llamada no es bloqueante. Mientras se espera la respuestas, se procesan otros mutantes         */
10     MutationArray mutationArray;
11     mutationArray , k, S'= sendPartitionsToSlaves(partitionIterators, S, M);
12     fixFound = mutationArray.fixFound();
13     S.addAll(S') ;                                                    // se guardan inputs que fallaron
14     M.feedBack= k ;                                  // se guarda el último valor de feedback calculado

15     if fixFound then
16         M'=M.applyMutsAndUpdatePendingMuts( mutationArray ) ;
17         return M'

18 if ! fixFound then
19     return "not fixable";
```

Algorithm 2: [Master server] distribución de particiones a los esclavos. Método "sendPartitionsToSlaves()"

Input: List < *LimitedArrayIterator* > partitionIterators, lista de iteradores limitados a la sentencia k
Input: S, set de inputs que satisfacen precondiciones de M y que exponen falla
Input: Method M, método a mutar sacado de la cola
Input: int k, número actual de sentencias que se pueden saltear (mismo k utilizado en los iteradores).
Input: freeSlaveServers, cola de servidores no ocupados.
Result: MutationArray mutArray, arreglo con secuencia de mutaciones para el método solución en caso de encontrarlo o "no fix found"
Result: k', resultado de getFeedBack en caso de que se encuentre un **k** mayor
Result: *List* < *Input* > S', conjunto de inputs obtenidos que exponen falla

```
1  int responseCount =0 ;                                   // cantidad de respuestas recibidas de los esclavos.
2  Thread thread2 ={
3  for ( LimitedArrayIterator iterator ∈ partitionIterators ) do
4  |   SlaveServer slave=freeSlaveServers.get();
5  |   slave.sendTask( iterator, S, M) ;
6  } .start();
7  int max_response_count= partitionIterators.size() ;
8  List < Input > S' ;                                      // conjunto de inputs recibidos que exponen falla
9  while (responseCount <max_response_count) do
   |   /* Recibe respuesta de un solo esclavo                                                          */
10 |   mutationArray,k',inputsList = receiveResponse() ;
11 |   responseCount ++ ;
12 |   S'.addAll ( inputList);
13 |   if (mutationArray.fixFound( )) then
14 |   |   return mutationArray,k',S';
   |   /* En un esclavo se detectó que se pueden saltear más sentencias ( se encontró un K mayor). Se frena todo lo que se
   |      estaba distribuyendo.                                                                       */
15 |   if (k' >k ) then
   |   |   /* Deja de enviar tareas de este lote especificado en partitionIterators.                 */
16 |   |   stopSending() ;
   |   |   /* Se cancelan las tareas enviadas de este lote a esclavos. Se trata de particiones en las que se utiliza el k
   |   |      anterior, el cual no sirve más.                                                         */
17 |   |   cancelTasks() ;
18 |   |   return "no fix found",k',S';
19 return"no fix found", k+1, S';
```

Algorithm 3: [Slave server] Pruning along mutations traversal

Input: LimitedArrayIterator limitedIterator;
Input: *List* < *Input* > I, inputs que exponen una falla obtenidos hasta el momento.
Input: Method M, método a mutar
Result: MutationArray mutArray, arreglo con secuencia de mutaciones para el método solución en caso de encontrarlo
Result: k, resultado de getFeedBack en caso de que se encuentre un **k** mayor
Result: *List* < *Input* > I', conjunto de inputs obtenidos que exponen falla

```
1  boolean candidateFound=false;
2  int k=limitedIterator.getKlimit();
3  while limitedIterator.hasNext() && ! candidateFound do
       /* Aplica mutación y decrementa en uno el limite de mutaciones para esa sentencia. En el vector de mutaciones,
          aumenta en uno en la posición de la mutación y si es necesario hace carry.                              */
4      mutArray = limitedIterator.next();
5      Method M'=M.applyMutsAndUpdatePendingMuts( mutArray );
6      boolean passesRac? = false;
7      for Input i ∈ I do
8          passesRac?= JML-RAC(M', i);
9          if ! passesRac? then
10             break ;
11     if passesRac? then
12         boolean passesTaco?= TACO(M');
13         if passesTaco? then
               /* Se necesita pasar RAC una vez más con todos los inputs encontrados hasta el momento antes de devolver
                  una solución. Esto se debe a que puede ocurrir que Taco no haya detectado un ciclo infinito.      */
14             booleanpassesLastRac?=lastRacAnalisisMaster(M') ;    // Se ejecuta en master con todos los contraejemplos
15             if passesLastRac? then
16                 return mutArray, k, I ;                                          // Solución encontrada

       /* Este if esta por claridad, si llego hasta aquí seguro se cumple la condición.                          */
17     if (! passesRac || ! passesTaco || ! passesLastRac) then
18         if ( M'.furtherMutationsAreAllowed() ) then
19             sendToMaster( M' );                    // Se envía al servidor maestro para aplicar mutaciones sucesivamente
20         Input input'= failing input from JML-RAC or TACO;
21         if ( ! I.contains(input') ) then
22             I.add(input');
23         k = getFeedBack(M',input');
24         if ( k <limitedIterator.limit() ) then
25             limitedIterator.setToZeroAllPositionsInRange(0, k-1);
26             limitedIterator.advanceOneStartingAtIndex(k);
27         else
28             return "fix not found", k,I;
29 return "fix not found", k,I;
```

4. Arquitectura

4.1. Arquitectura de aplicación

La arquitectura de la aplicación es de 2 capas: master-slaves. Se cuenta con un servidor maestro que distribuye tareas a los servidores esclavos y recibe los resultados. Las tareas que requieren procesamiento intensivo son realizadas en los esclavos. En el servidor maestro se analizan los resultados provenientes de los esclavos y se determina si se encontró la solución o si hay que continuar buscando. Los datos se recolectan en el servidor maestro y a cada esclavo solamente se le envía la información necesaria para ejecutar la tarea que se le asigna.

Figura 2: Diagrama UML de despliegue.

4.2. Controllers

La versión original de la aplicación Stryker está formada por controladores, los cuales son módulos que contienen principalmente tres elementos:

1. Una cola de inputs que representa a las tareas que debe procesar dicho controlador. Estos inputs provienen de otros controladores principalmente.
2. Un thread que toma un input de la cola y que ejecuta la tarea que se corresponde con ese input.
3. Un flag que indica si se está apagando.

Estos tres elementos se encuentran en una clase abstracta llamada AbstractBaseController de la cual heredan todos los controladores originales; estos son: DarwinistController, MuJavaController, OpenJMLController, UnskippableMuJava-Controller.

En la versión distribuida, se adaptaron los controladores originales de Stryker y se agregaron otros de forma tal que se posibilite el funcionamiento distribuido. Durante el desarrollo fue necesario abstraer la obtención de controladores para obtener la implementación que corresponde según el tipo de servidor que se esta corriendo (maestro o esclavo). Por ejemplo, en MuJavaControllerSlave cuando se necesita enviar un mutante para que se realice un análisis de RAC se debe encolar esta tarea al OpenJMLControllerSlave y no al OpenJMLControllerMaster. Con esta idea en mente, se agregó un método en la clase AbstractBaseController para obtener los controladores que correspondan: en los controladores originales este método devuelve el conjunto de controladores originales, en los controladores del servidor maestro se obtienen la versión maestra de los controladores y en los controladores del servidor esclavo se devuelven la versión esclava de dichos controladores. Esto se puede apreciar en el diagrama de herencia que se encuentra a continuación (figura 3). En el mismo,"ControllerX" representa genéricamente a algún controlador original de Stryker, "ControllerXMaster" representa a la especialización del controlador "ControllerX" para el servidor maestro, y "ControllerXSlave" lo mismo pero para el servidor esclavo.

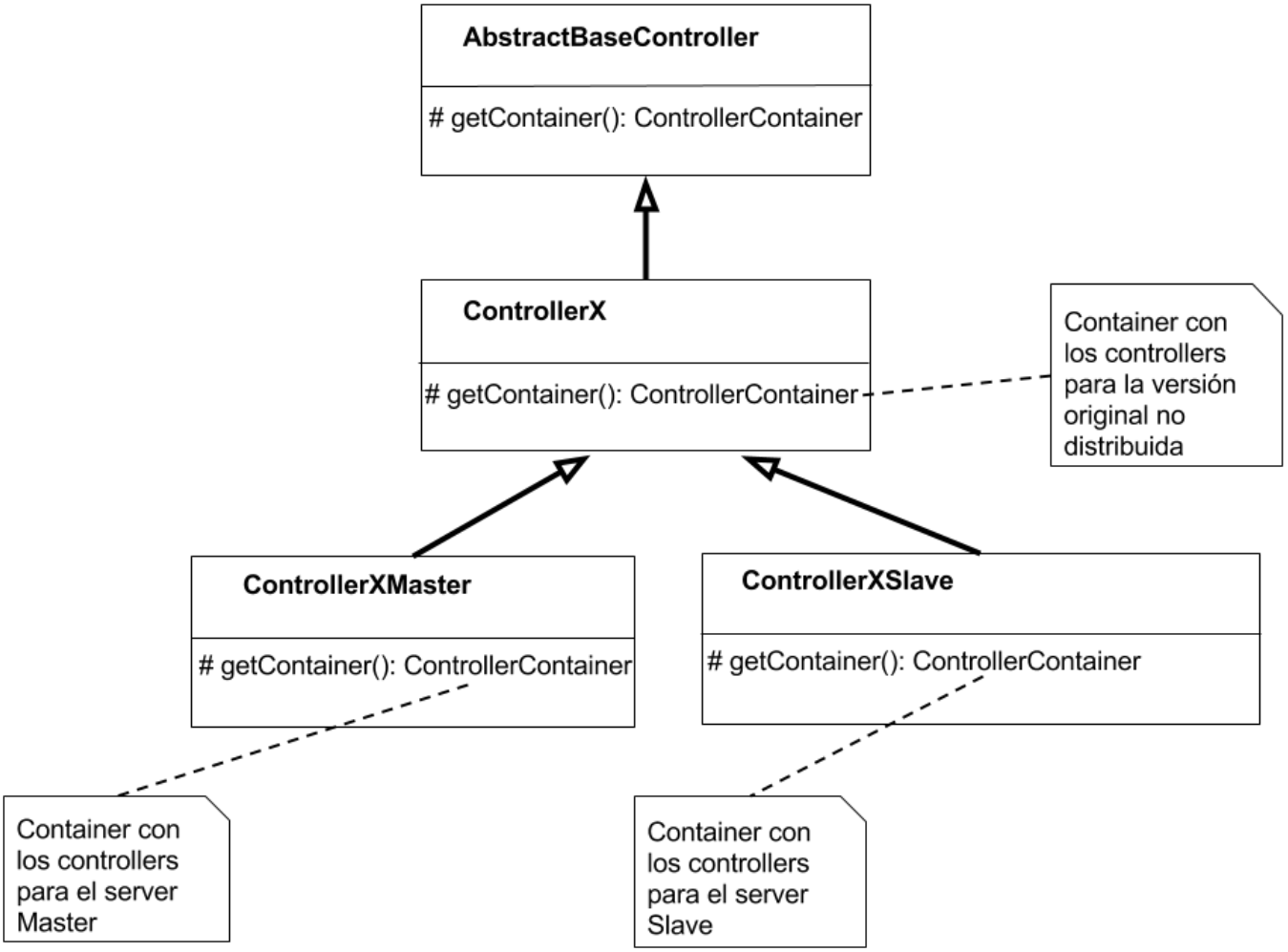


Figura 3: Herencia de controllers.

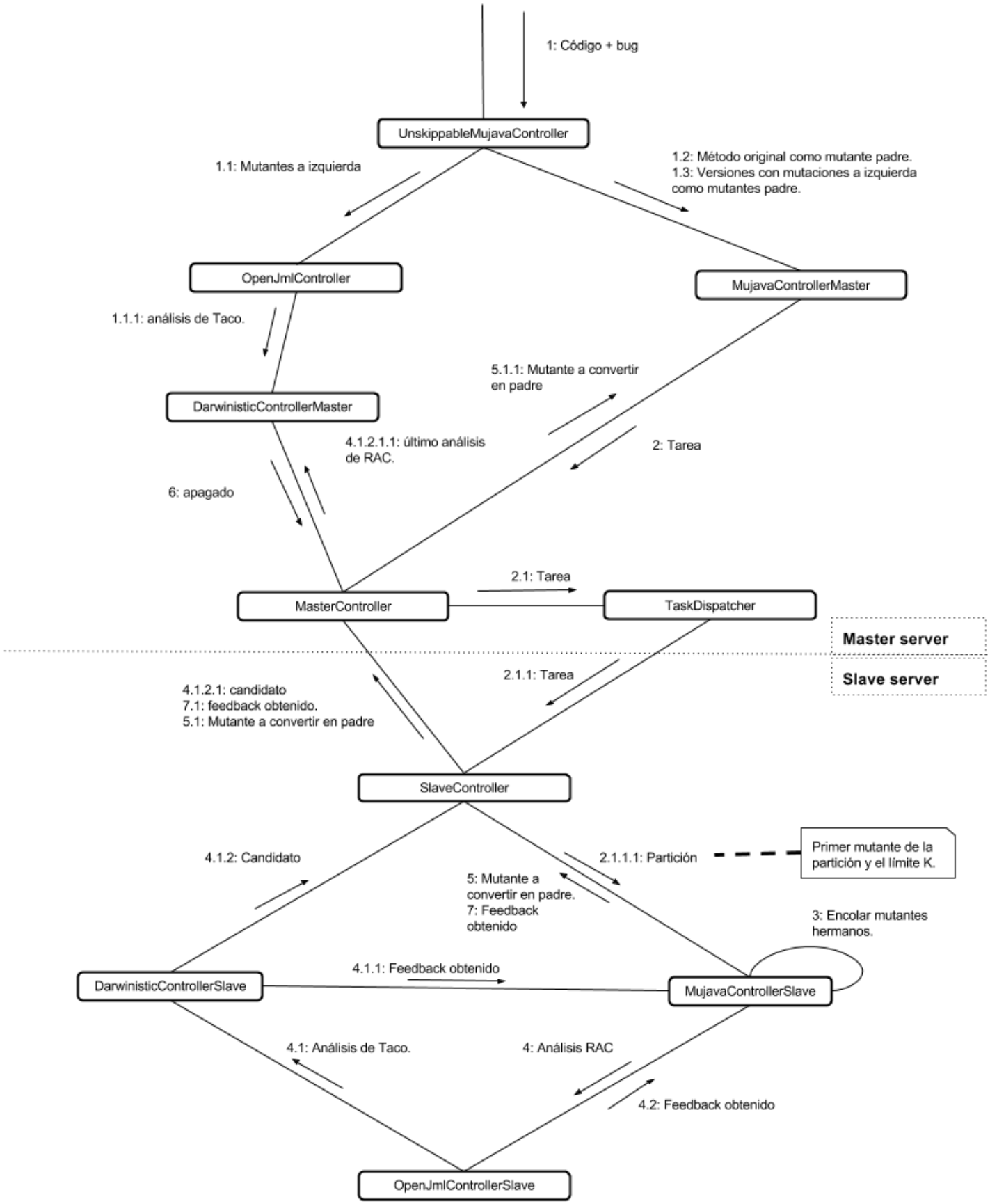


Figura 4: Diagrama UML que representa a la comunicación entre controladores.

4.2.1. Controllers esclavos

Del lado esclavo los controladores que se tienen se pueden apreciar en la parte inferior de la figura 4 y son los siguientes:

- **SlaveController**. Toda la comunicación entre el servidor esclavo y el maestro pasa por este controlador. Recibe la partición a trabajar desde el maestro y se la asigna al MuJavaControllerSlave. También envía al servidor maestro el resultado de procesar una partición. Por último realiza el inicio/apagado/reinicio del servidor esclavo; las últimas dos operaciones las realiza cuando se lo indica el servidor maestro y la primera cuando se inicia la aplicación esclavo.

  A diferencia de los otros controladores del servidor esclavo, no extiende a la clase AbstractBaseController. Sin embargo su funcionamiento es similar: contiene un ExecutorService que es parecido a un thread al cual se le encolan tareas ejecutables que consisten a su vez en encolarle inputs al MuJavaControllerSlave provenientes del servidor maestro.
- **MuJavaControllerSlave**. Recibe la partición que viene del maestro y explora las mutaciones dentro de dicha partición. Posteriormente, se encarga de responder al maestro el resultado de explorar la partición. Este resultado puede ser el nuevo valor de feedback o una indicación de que terminó sin resultados. En todos los casos que le responde al servidor maestro, lo realiza a través del SlaveController (actúa como proxy) y además le envía los contraejemplos obtenidos mientras trabajó en dicha partición.
- **DarwinistControllerSlave**. Utiliza Taco para calcular el valor "getFeedback()" y para validar candidatos provenientes del OpenJMLController que pasaron el análisis RAC. En caso de que el mutante candidato también pase el análisis de Taco, envía esta posible solución al DarwinistControllerMaster que se encuentra en el servidor maestro para ejecutar un último análisis de RAC. Adicionalmente, envía los contraejemplos obtenidos en esta respuesta. La respuesta es enviada al maestro a través del SlaveController.
- **OpenJMLController**. Recibe un mutante y realiza un análisis de RAC para descartar rápidamente a los mutantes que no son solución. Durante este análisis utiliza los contraejemplos recibidos del maestro al momento de recibir la partición junto con los generados durante los análisis de Taco ejecutados en dicho esclavo (en el DarwinistControllerSlave). Si el mutante recibido no pasa el análisis de RAC, se lo encola al MuJavaControllerSlave para que siga explorando la partición a partir del siguiente mutante.

El funcionamiento general inicia cuando MuJavaControllerSlave toma un mutante de la cola, se lo envía al OpenJMLController para validarlo. Si es válido pasa al DarwinistControllerSlave para validar con Taco y si pasa esto último se envía al servidor maestro como candidato. Si falla RAC en OpenJMLController o si falla Taco en DarwinistControllerSlave, se encola en MuJavaControllerSlave para generar el siguiente mutante. La generación del siguiente mutante en MuJavaControllerSlave necesita llamar al DarwinistControllerSlave para computar getFeedback. Una vez que tiene este dato que utiliza para podar mutantes, obtiene el siguiente mutante relevante y finalmente se lo envía al OpenJMLController para continuar con el ciclo anterior.

4.2.2. Controllers maestros

En el servidor maestro se tienen los siguientes controladores que se pueden visualizar en la parte superior de la figura 4:

- **MuJavaControllerMaster**. Inicialmente recibe en su cola el método original y versiones del mismo con mutaciones a izquierda considerados como casos padre. Toma un mutante de la cola, el mismo puede ser el primer hijo de un padre junto con la indicación de feedback o un mutante para convertir en padre. En el último caso convierte el mutante en padre y obtiene el primer hijo. En ambos casos, usando el primer hijo y el último valor de feedback obtenido genera las particiones que corresponden. Las particiones generadas se terminan encolando al TaskDispatcher.
- **DarwinistControllerMaster**. La primer funcionalidad que realiza es la de ejecutar el último análisis de RAC para validar un mutante candidato. Si pasa este último análisis de RAC, la segunda funcionalidad que realiza es la de notificar al MasterController que se debe apagar el clúster.
- **UnskippableMuJavaControllerMaster**. Realiza la misma funcionalidad que en la versión original de Stryker. Básicamente recibe el código de un método con bugs, luego le encola al MuJavaControllerMaster los mutantes padres iniciales y después le encola al OpenJMLControllerMaster los mutantes con mutaciones a izquierda que genera inicialmente.
- **OpenJMLControllerMaster**. Realiza análisis de RAC sólo para los mutantes con mutaciones a izquierda generados por el UnskippableMuJavaControllerMaster inicialmente. Los demás análisis de RAC (los que se necesitan para evaluar mutantes dentro de cada partición) se realizan en el esclavo al que le corresponde dicha partición; la idea de esto es delegar esta tarea al esclavo y de esta manera alivianar la carga del servidor maestro.
- **TaskDispatcher**. Levanta una partición a enviar de una cola y luego de otra cola extrae una referencia a un servidor esclavo que no se encuentre ocupado, y le termina enviando la tarea (partición a analizar) a dicho servidor esclavo. Adicionalmente, para mejorar la performance evita enviar particiones obsoletas de un determinado padre, lo cual ocurre cuando el valor de kLimit (sentencia a la que está limitada la partición) es menor al que actualmente se está utilizando en ese padre.

- **MasterController**. Todos los controladores del lado maestro (a excepción del controlador TaskDispatcher) que se comunican con los servidores esclavos lo hacen a través de este controlador. Este controlador realiza tres tareas importantes para el funcionamiento del programa en su conjunto. La primera consiste en recibir los resultados provenientes de los esclavos. Entre las respuestas, se encuentran los mutantes a convertir en padres y los mutantes candidatos. La segunda tarea es la de registrar a los esclavos que están libres para recibir nuevas particiones. La tercera consiste en la cancelación de particiones que son obsoletas en los esclavos; esto ocurre cuando se detectó que esas particiones se pueden podar. Otra funcionalidad que realiza es la notificar a cada esclavo que debe apagarse ya que se encontró la solución. A diferencia de los otros controllers del lado maestro, este no extiende a la clase AbstractBaseController. Internamente, tiene una cola de mutantes candidatos los cuales debe compilar en el maestro y luego encolarlos en el DarwinistControllerMaster para realizarles el último análisis de RAC.

4.3. Mensajes

Los datos que se envían entre el servidor maestro y los esclavos se encapsulan en un objeto al se llamó *mensaje*. Se tiene una clase abstracta de mensajes llamada MessageWrapper donde se implementa la lógica para pasar archivos entre servidores y se almacena un mapeo para reemplazar las referencias a los archivos en el filesystem de origen con el nuevo path de dicho archivo en el servidor destino. De esta clase abstracta heredan dos clases. Una se llama SlaveInputWrapper y se utiliza para modelar particiones que se envían al servidor esclavo. La otra clase se llama SlaveResponseWrapper y se emplea para representar respuestas del servidor esclavo dirigidas al servidor maestro.

Hay tres subclases de SlaveResponseWrapper. La primera es CandidateMessage y se utiliza para encapsular un mutante candidato. La segunda clase es MuJavaResponseMessage, la cual se emplea para indicar que se agotó la partición asignada al esclavo. La tercera clase es FatherizeMessage, la cual representa un mutante a convertir en padre en el servidor maestro.

Adicionalmente, en cada mensaje de respuesta que genera el servidor esclavo se incluyen los datos de las estadísticas de Stryker recolectados desde la última vez que se envió una respuesta hasta la respuesta en cuestión.

4.4. Comunicación Master-Slave

Para realizar la comunicación entre el servidor maestro y los servidores esclavos se utilizó JRMI (Java Remote Method Invocation). Se tienen dos objetos remotos, uno en el servidor maestro llamado MasterController y otro en los servidores esclavo denominado SlaveController. El servidor maestro exporta su objeto MasterController a un servidor de registro RMIRegistry; los servidores esclavos obtienen una referencia del objeto remoto MasterController al conectarse al servidor RMIRegistry. Luego, cada servidor esclavo le envía un mensaje al servidor maestro usando la referencia al MasterController que posee, en ese mensaje le pasa una referencia del objeto remoto SlaveController. Entonces, el servidor maestro obtiene una referencia del SlaveController de cada esclavo que se haya conectado, y también cada servidor esclavo posee una referencia del servidor maestro. Todos los mensajes que se envían entre servidor maestro y servidores esclavos en ambas direcciones se realizan utilizando estos objetos remotos.

Al utilizar este modelo se abstraen las capas inferiores de comunicación, y se resuelven los problemas de conexión y envío de mensajes entre servidores. Solamente se invoca el método correspondiente en el objeto remoto pasándole como parámetro el mensaje que se le envía al otro servidor.

Asimismo, al utilizar JRMI, todos los parámetros de los métodos que se pueden invocar en el objeto remoto deben ser serializables. Por este motivo todos los mensajes mencionados en la sección anterior son serializables.

5. Funcionamiento

5.1. Encendido

Cuando se inicia el servidor maestro, primero se ejecuta un análisis de TACO para confirmar que la clase a reparar contiene bugs, si es así, primero se inicia el servidor de registro RMIRegistry y se exporta el objeto remoto MasterController. Luego se inician los controladores maestros y el servidor de autodiscover. Este último escucha en la red esperando por un paquete broadcast UDP enviado por un servidor esclavo para responderle con la dirección IP del servidor maestro. Después, se envía la clase a reparar al controlador maestro UnskippableMuJavaControllerMaster, donde se comienzan a aplicar mutaciones.

Al encender el servidor esclavo, se inicia la función de autodiscover para obtener la dirección IP del servidor maestro. Luego se obtiene la referencia remota al MasterController del servidor de registro. Después se inician los controladores esclavos y se registra en el servidor maestro indicando que se encuentra disponible para procesar tareas.

5.2. Solución

Cuando en un servidor esclavo se encuentra un mutante que pasa el análisis de RAC y el análisis de TACO, es decir, un mutante candidato, el mismo se envía desde el controlador esclavo DarwinistControllerSlave al controlador maestro DarwinistControllerMaster para realizar un último análisis de RAC utilizando todos los contraejemplos que posee el servidor maestro. Para llegar hasta allí, debe pasar por el SlaveController en el servidor esclavo que lo envía al servidor maestro invocando un método en la referencia al objeto remoto MasterController que posee. Una vez realizado el último análisis de RAC en el servidor maestro, si pasó dicho análisis, el mutante es una solución, en caso contrario se descarta. Si el mutante es solución, el controlador DarwinistControllerMaster notifica al MasterController para que realice el apagado.

5.3. Apagado

Como se mencionó en la sección anterior, cuando se encuentra una solución en el DarwinistControllerMaster, se le notifica al MasterController para que realice el apagado. El MasterController le envía la notificación a todos los esclavos que se hayan conectado, y luego inicia un thread para apagar todos los controladores maestros y el servidor RMIRegistry. En cada servidor esclavo que recibe la notificación, también se inicia un thread que realiza el apagado de todos los controladores del lado esclavo y lleva a cabo el unexport del objeto remoto para que deje de estar disponible.

5.4. Reinicio de esclavos/cancelación de tareas

Cuando se procesan las particiones en los servidores esclavos, puede ocurrir que se encuentre un valor de feedback que permite podar mutaciones que están por fuera de esa partición, es decir en otra, y que posiblemente esa otra partición se esté procesando en otro servidor esclavo. Entonces cuando en el servidor maestro, el controlador MuJavaControllerMaster recibe una respuesta que indica esto, le notifica al controlador maestro MasterController que se deben reiniciar los servidores esclavos que estaban procesando particiones que se pueden podar por este nuevo valor de feedback que indicaba la respuesta. El MasterController le notifica a los servidores esclavos que deben ser reiniciados. En cada uno de estos servidores, si siguen procesando esa partición, se inicia un thread que primero realiza el apagado de todos los controladores esclavos, y luego los vuelve a encender. Al finalizar, los servidores esclavos le notifican al servidor maestro que se encuentran preparados para recibir nuevas tareas.

6. Dificultades encontradas

Durante el desarrollo de la solución presentada en este informe se tuvieron que afrontar múltiples dificultades, entre las que se encuentran:

- Fallas en el código original de Stryker y de Taco. Estas fallas tuvieron que ser atendidas para poder ejecutar el conjunto de casos de prueba.
- La principal falla detectada en el código original de Stryker consiste en que para una misma prueba en múltiples ejecuciones, puede ocurrir que se obtenga una misma solución o que se la omita directamente. Debido a esto en varias pruebas se necesitaron múltiples corridas para reproducir determinada situación que se estaba evaluando y para obtener los resultados finales.
- El código original es complejo y no cuenta con documentación escrita sobre la implementación. Para esto, se debió consultar directamente a los autores de dicho código.
- El código original de Taco y Stryker se debe correr sobre Java JDK 1.7 en forma exclusiva, no permite utilizar versiones más nuevas. Esto complicó el establecimiento del entorno de trabajo.
- Durante el desarrollo se detectó el no determinismo de la herramienta MuJava que es utilizada por el código original de Stryker. Esto dificultaba reproducir determinadas situaciones de falla durante la implementación. Debido a esto, se debió contactar al autor de dicha herramienta para solucionar este problema.

7. Evaluación

Los resultados que se van a mostrar a continuación en esta sección fueron obtenidas utilizando hasta 16 computadoras modelo "Dell optiplex 9020 ". Las mismas se encuentran equipadas con procesador "Intel(R) Core(TM) i7-4790 CPU" corriendo a 3.60GHz (sin overclocking), 8 GB de RAM. El sistema operativo utilizado es GNU/Linux 4.9.40-1-lts en distribución ArchLinux. Además, para realizar pruebas con hasta 32 computadoras se agregaron otras 16 pc modelo "Dell optiplex 790 ", las cuales tienen un procesador "Intel(R) Core(TM) i7-2600 CPU" corriendo a 3.40GHz (sin overclocking), y con igual tamaño de memoria y sistema operativo.

Stryker distribuido fue ejecutado como un jar ejecutable de Java, utilizando OpenJDK 1.7 como la plataforma Java subyacente. A los experimentos en los casos de prueba se les asignó un límite de tiempo de 10 horas.

7.1. Especificación de casos de prueba

Se utilizaron las pruebas que se encuentran en *workspaceStryker/comitaco/tests/icse/* dentro del directorio que contiene el código de todo el proyecto. Estas pruebas pertenecen al proyecto de Stryker original y fueron creadas para probar la herramienta. En total son 4 clases Java de estructuras de datos que se prueban. La primer clase es SinglyLinkedList: es una implementación de lista simplemente encadenada con los métodos *contains*, *getNode*, e *insert*. La segunda clase es NodeCachingLinkedList: es una lista doblemente enlazada y circular, que implementa la interfaz *List* del paquete "Apache Commons collections" con métodos *contains*, *insert* y *remove*. La tercera clase es BinarySearchTree: es un árbol de búsqueda binaria con métodos *contains*, *insert*, y *remove*. La cuarta clase es BinomialHeap: es una implementación de colas de prioridades utilizando heap binomial con métodos $findMin$, *insert*, $extractMin$. En cada clase se prueban los 3 métodos mencionados y por cada uno de estos últimos hay 5 test generados de 1,2,3 y 4 bugs. En cada test se prueba un método y se agregan cierta cantidad de bugs a la clase original según se indica en el nombre de cada clase Java de prueba.

7.2. Resultados experimentales

Para comparar los tiempos de ejecución entre el tiempo de correr en 1 pc o en el clúster, se calculó el "speed up"(cantidad de veces más rápido) con la siguiente fórmula:

$$SpeedUp = \begin{cases} \frac{-Tiempo_{distribuida}}{Tiempo_{monolítica}} + 1 & \text{si } Tiempo_{distribuida} > Tiempo_{monolítica} \\ \\ \frac{Tiempo_{monolítica}}{Tiempo_{distribuida}} - 1 & \text{otro caso} \end{cases} \quad (2)$$

Advertencia: en los siguientes gráficos, cuando se usan 32 pc se están agregando 16 computadoras más que son 2 generaciones anteriores que las primeras 16 que se usan en las pruebas de hasta 16 pc. Debido a esto, la diferencia entre las curvas de 16 y 32 pc debería ser más marcada si se utilizase máquinas idénticas.

7.2.1. Peores casos de la versión distribuida

Para cada clase se tomaron los 2 casos que dieron los valores de speedUp (ver fórmula 2) más bajos en la versión distribuida utilizando 16 pc contra la ejecución en una sola pc. En el siguiente cuadro se muestran los resultados:

Test	Tiempo 1 pc [ms]	Tiempo 16 pc [ms]	SpeedUp 16	Tiempo 32 pc [ms]	SpeedUp 32
icse.binomialheap.set5.StrykerBinomialHeapExtractMin2Bug12Dx26DTest	358862	460712	-0.283814	497360	-0.385937
icse.binomialheap.set2.StrykerBinomialHeapExtractMin2Bug17Ix26DTest	584903	916828	-0.567487	469847	0.244880
icse.bintree.set2.StrykerBinTreeRemove4Bug6Dx18Dx32Dx44DTest	139343	152340	-0.093273	70964	0.963573
icse.bintree.set5.StrykerBinTreeRemove3Bug21Ix26Dx40ITest	138762	147234	-0.061054	114179	0.215302
icse.nodecachinglinkedlist.set1.StrykerNodeCachingLinkedListAddFirst1Bug4DTest	10743	36872	-2.432188	10119	0.061666
icse.nodecachinglinkedlist.set2.StrykerNodeCachingLinkedListAddFirst2Bug4Ix6ITest	7081	7094	-0.001836	6580	0.076140
icse.singlylinkedlist.set1.StrykerSinglyLinkedListContains2Bug4Dx10DTest	13651	15961	-0.169218	20615	-0.510146
icse.singlylinkedlist.set4.StrykerSinglyLinkedListGetNode1Bug9ITest	4716	4742	-0.005513	4705	0.002338

Cuadro 1: Tabla de resultados para peores casos.

Para cada clase y cantidad de pc, se acumularon los tiempos de ejecución de estos peores casos. Adicionalmente, se corrieron estos mismos casos agregando 16 computadoras (teniendo un total de 32) para ver cómo se comporta la aplicación cuando se la escala en los mismos casos.

En la figura 5 se muestra un gráfico de barras, donde las barras representan la suma de los tiempos de la corrida de los dos peores tests en cada clase, en la parte superior de la barra se indica el speedup, cada barra de un color distinto representa la cantidad de computadoras usadas en esa corrida, y en el eje horizontal se muestra el nombre de la clase testeada.

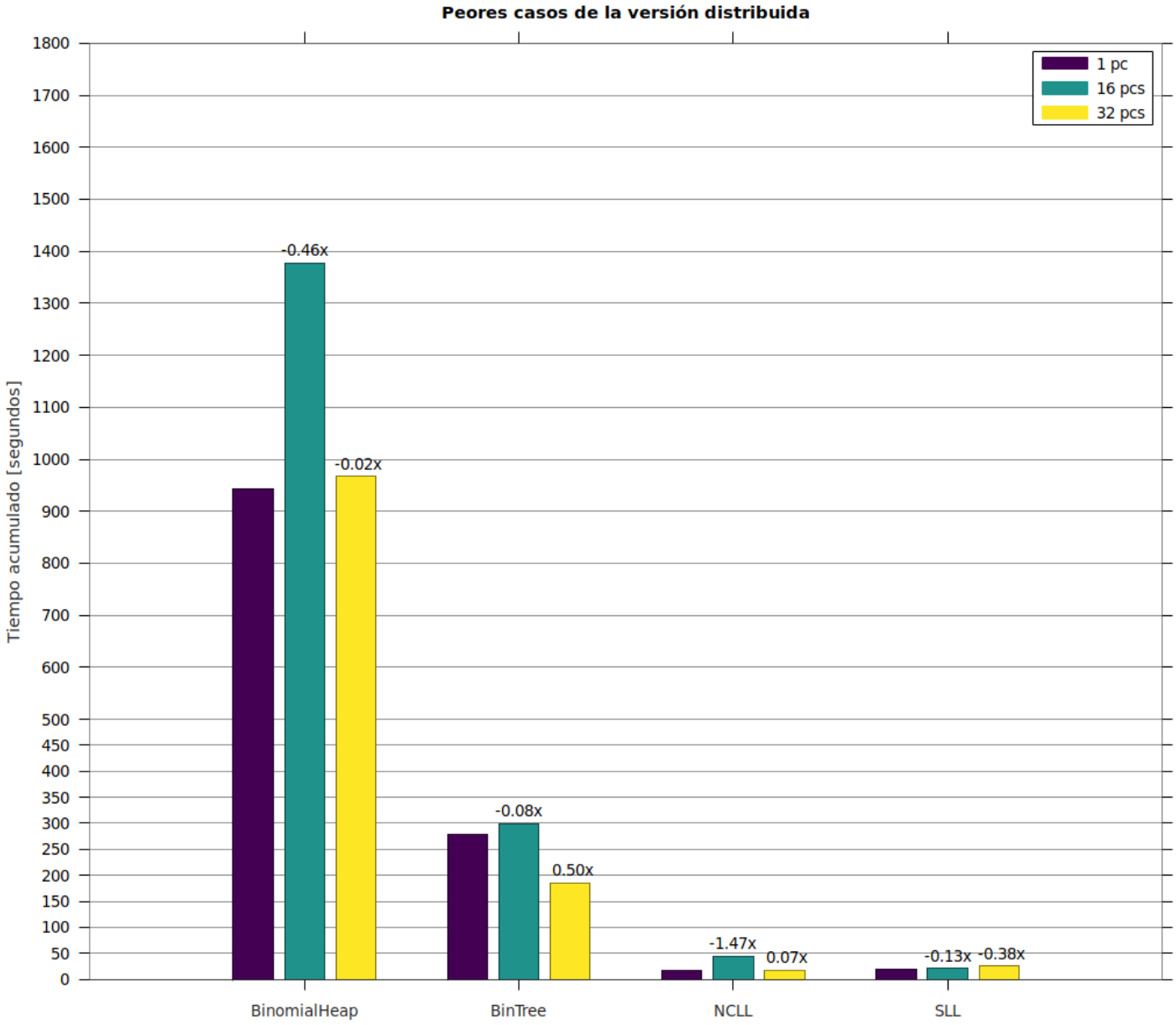


Figura 5: Gráfico de barras para los peores casos de la versión distribuida.

En la figura 5 se puede apreciar que en los peores casos para la corrida con 16 pc se llegó a alcanzar un incremento de tiempo de 1,47 veces al acumular el tiempo para los dos casos de cada clase. Para la corrida con 32 máquinas en cambio, en los peores casos no hay diferencia significativa con correrlo en una sola pc e incluso se obtuvo una leve mejora respecto con la ejecución en una sola pc.

7.2.2. Mejores casos de la versión distribuida

Para cada clase se tomaron los 2 casos en los que se obtuvieron los mayores valores de speedUp en la versión distribuida en 16 pcs contra la corrida en 1 pc. Por cada clase, los resultados en los tests de mejores casos son los siguientes:

Test	Tiempo 1 pc [ms]	Tiempo 16 pc [ms]	SpeedUp 16	Tiempo 32 pc [ms]	SpeedUp 32
icse.binomialheap.set4.StrykerBinomialHeapExtractMin4Bug9Dx23Dx30Dx38DTest	10345736	1324976	6.808244	789767	12.099732
icse.binomialheap.set3.StrykerBinomialHeapFindMinimum4Bug2Dx3Dx6Dx11DTest	1681355	141580	10.875653	127096	12.229016
icse.bintree.set1.StrykerBinTreeRemove3Bug14Dx37Dx38ITest	1378891	66701	19.672719	47488	28.036620
icse.bintree.set1.StrykerBinTreeRemove4Bug3Dx6Dx17Dx43ITest	5179328	190334	26.211786	309225	15.749383
icse.nodecachinglinkedlist.set2.StrykerNodeCachingLinkedListContains4Bug1Dx2Dx7Dx8DTest	536027	35316	14.178021	43472	11.330397
icse.nodecachinglinkedlist.set3.StrykerNodeCachingLinkedListContains3Bug3Dx5Dx7DTest	493361	16539	28.830159	24219	19.370825
icse.singlylinkedlist.set5.StrykerSinglyLinkedListGetNode4Bug5Dx6Ix8Dx11DTest	1581820	109900	13.393267	30306	51.194945
icse.singlylinkedlist.set4.StrykerSinglyLinkedListGetNode4Bug4Dx5Dx8Dx9DTest	2196953	41239	52.273673	39809	54.187345

Cuadro 2: Tabla de resultados para mejores casos.

En la tabla 2 se puede apreciar que la última prueba presenta un speed up excepcional de 52. Para este caso, la

solución se encuentra en la partición 17 del kLimit 3 para el primer padre. El lote al que pertenece esta partición contiene particiones que consumen mucho tiempo cada una debido a que requieren que se recorra muchos mutantes. La importante disminución de tiempo que se puede apreciar al correrlo en múltiples computadoras se debe a que cuando se llega a este lote, se trabajan en paralelo las particiones y de esta manera se llega al resultado sin tener que recorrer cada partición por completo. Cuando este mismo caso se lo corre en una sola pc, se debe procesar secuencialmente a las 16 particiones del lote en forma completa antes de llegar a la partición 17 que contiene la solución; debido a esto se tarda mucho más al correr este caso en forma monolítica.

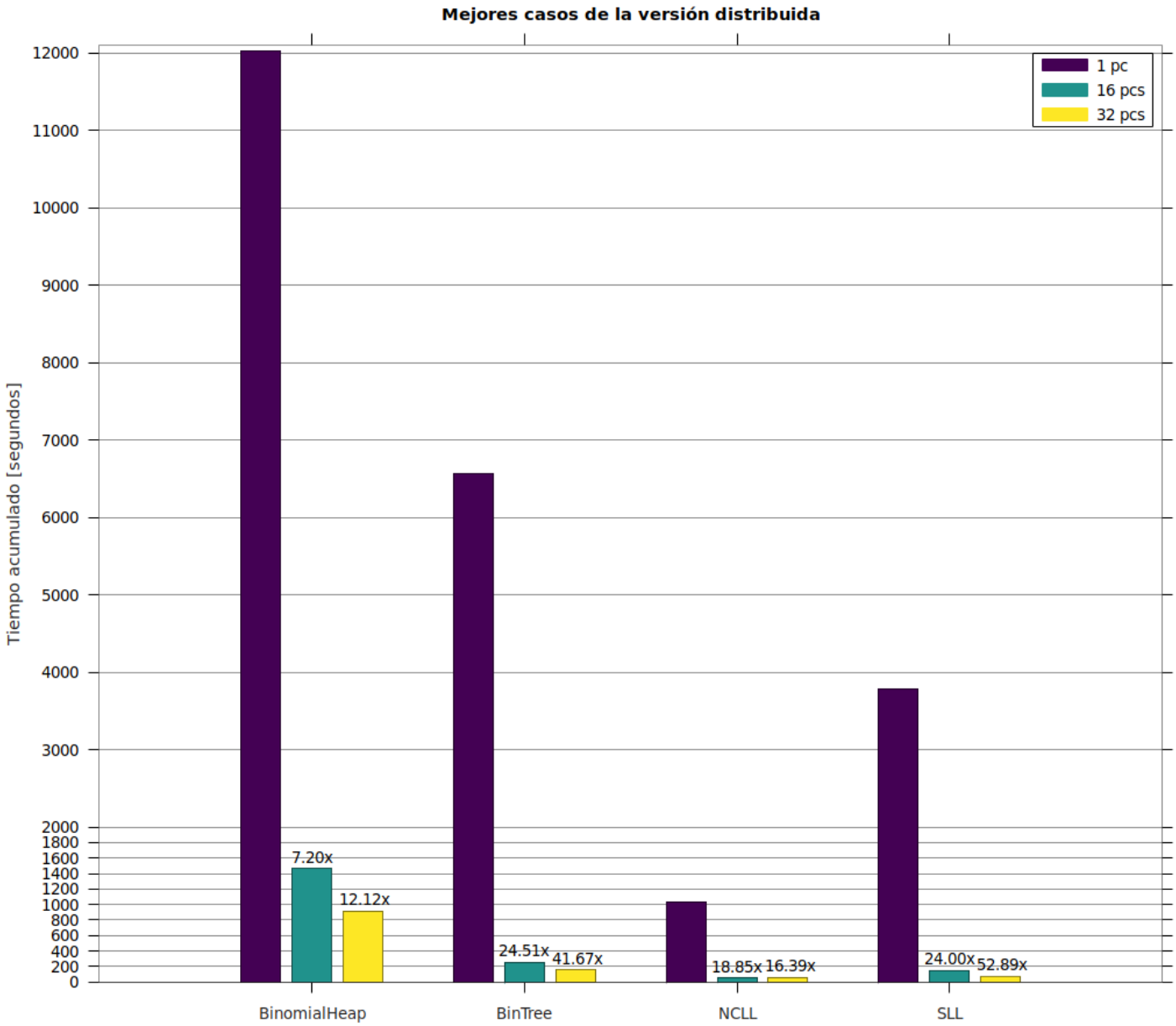


Figura 6: Gráfico de barras para los mejores casos de la versión distribuida.

En la figura 6 se muestra un gráfico de barras, donde las barras representan la suma de los tiempos de la corrida de los dos mejores tests para cada clase, en la parte superior de la barra se indica el speedup. Cada barra de un color distinto representa la cantidad de computadoras usadas en esa corrida, y en el eje horizontal se muestra el nombre de la clase testeada. En estos mejores casos hay una mejora significativa respecto de la ejecución monolítica, alcanzándose un speedup promedio de 52,89 en la clase SinglyLinkedList con 32 computadoras. En todas las clases excepto en una se obtuvieron menores tiempos al utilizar más máquinas. La excepción fue la clase NodeCachingLinkedList en la cual el tiempo para la ejecución con 32 máquinas fue levemente mayor que al utilizar 16 computadoras; esto se debe a que para problemas que se resuelven en poco tiempo (menos de un minuto) como en este caso, el mayor overhead necesario al trabajar con más computadoras se vuelve más notable.

7.2.3. Comparación peores y mejores casos

En los casos en los que se obtuvieron los mejores speed ups se requirió de más tiempo de ejecución en comparación con los peores casos dentro de cada clase. Esto quiere decir que el algoritmo corriendo en forma distribuida se ve beneficiado en las pruebas que requieren más tiempo de ejecución por requerir que se analicen mayor cantidad de mutantes para encontrar la solución o porque la clase es más compleja (es más costoso cada análisis de Taco). Por otra parte, los peores casos son aquellos en los que se requiere poco tiempo de ejecución en una sola pc y entonces el overhead de distribuir mutantes es más notorio.

7.2.4. Tiempos promedios

Se corrieron todas las pruebas variando la cantidad de computadoras: en 1 computadora (i7-4790), en 16 computadoras (i7-4790), y en 32 (se agregaron 16 i7-2600). Con estos valores se calculó el tiempo promedio en encontrar la solución para cada cantidad de bugs. En la figura 7 se puede ver que a medida que aumenta la cantidad de bugs, aumenta en forma exponencial el tiempo promedio que toma en encontrar la solución para las tres cantidades de computadoras. La pendiente en los casos de 16 y 32 pc es mucho menor en comparación con una sola pc; esto es todavía más pronunciado en los casos de 3 y 4 bugs donde en promedio la corrida es 7 y 8 veces menor en tiempo para 16 y 32 pc respectivamente. De esta manera, tiene más sentido utilizar la versión distribuida en pruebas de 3 o más bugs ya que en estos casos la disminución de tiempo es más notoria.

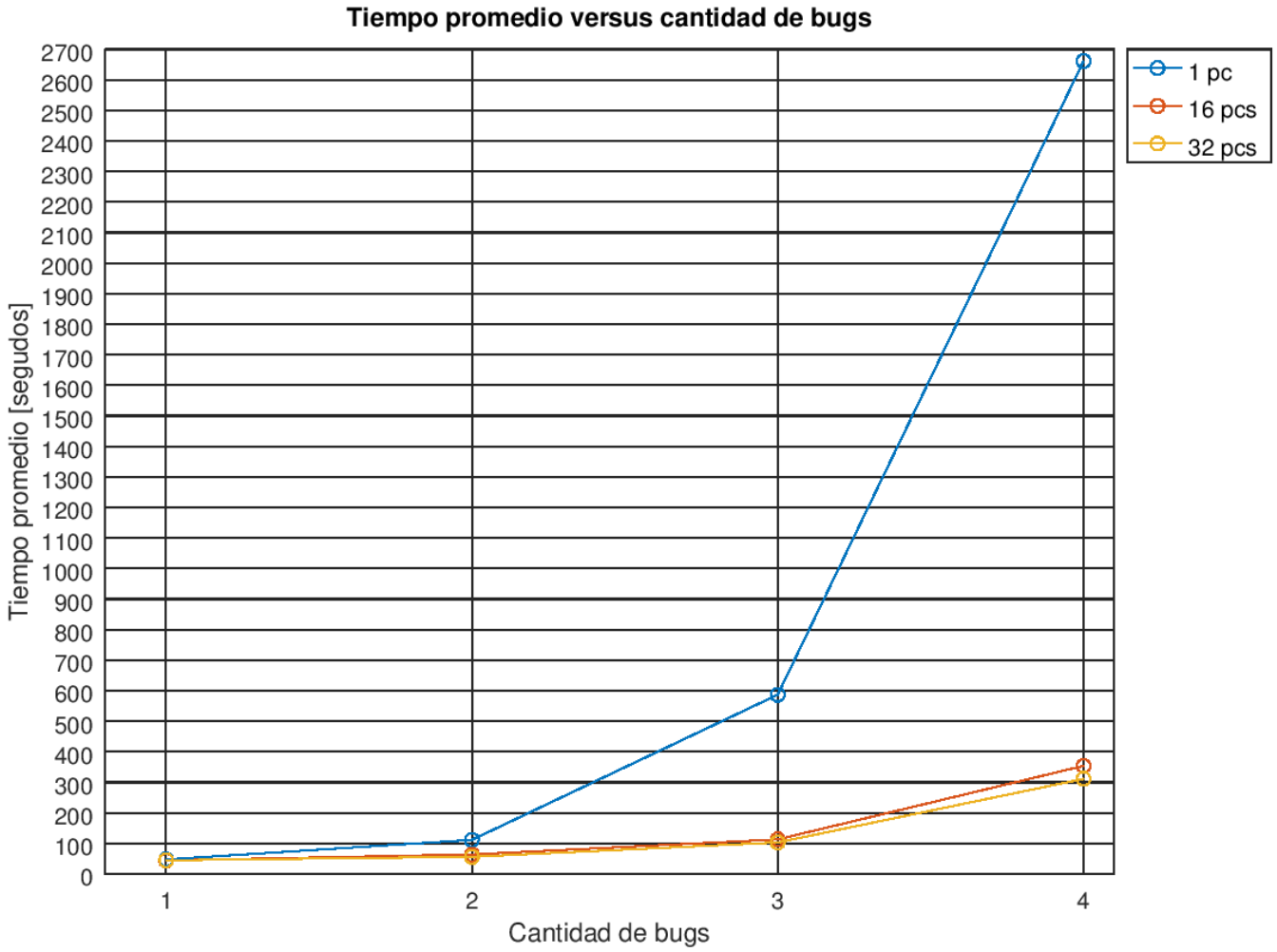


Figura 7: Promedio de tiempos según la cantidad de pc.

En el gráfico anterior (figura 7) no se distingue la diferencia entre los promedios para 16 o 32 pc.

Haciendo un acercamiento en la figura 7, se obtiene figura 8 donde se puede notar mejor la diferencia entre 16 y 32 pc. En 32 hay tiempos más bajos, pero la diferencia comienza a ser notable a partir de 4 bugs. De esta manera, se justifica utilizar 32 pc en lugar de 16 pc a partir de los 4 bugs.

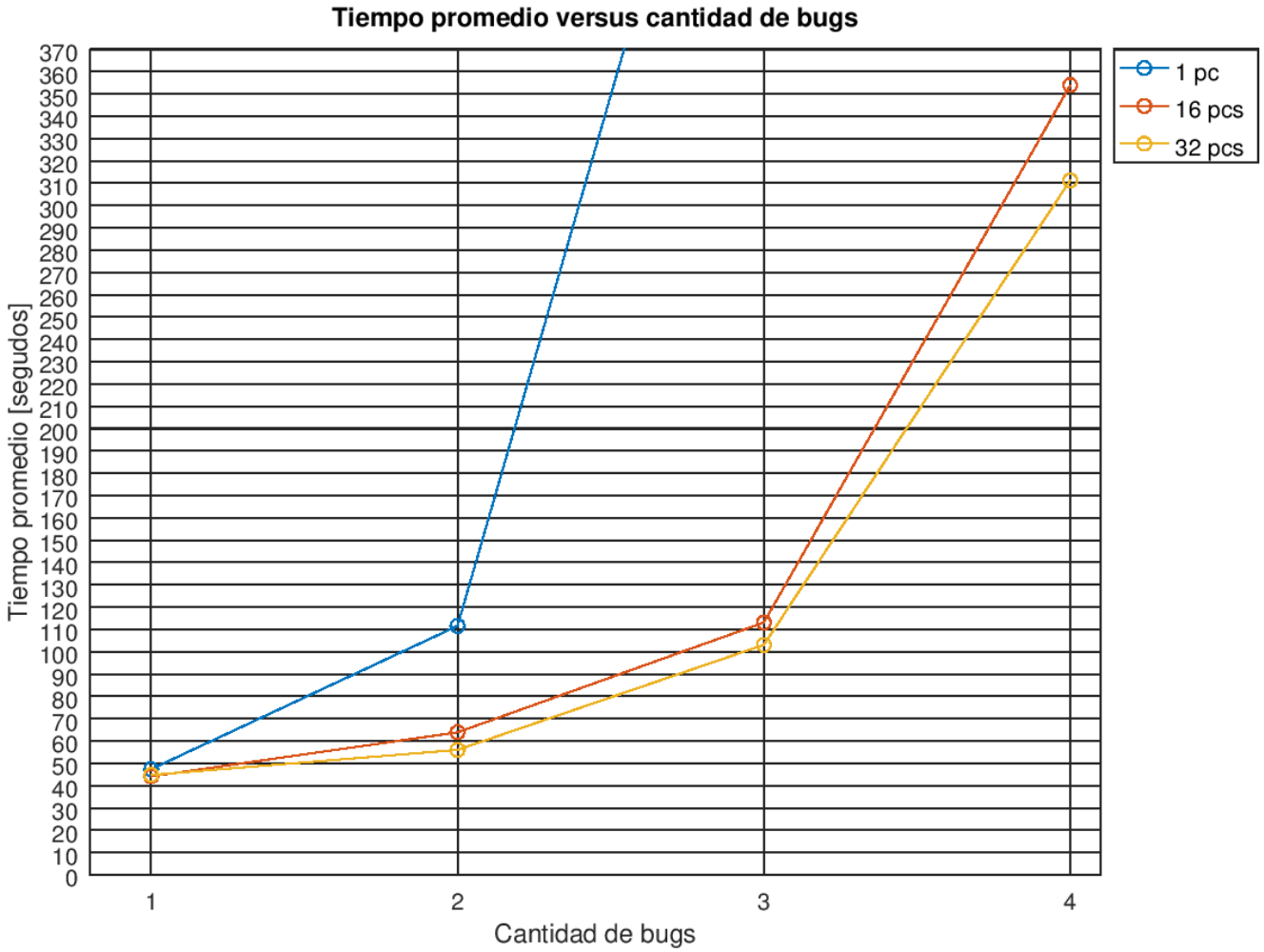


Figura 8: Promedio de tiempos según la cantidad de pc, acercamiento hasta 370 segundos.

7.2.5. Distribución de speed ups

En la figura 9 se graficó un boxplot para cada cantidad de bugs donde se ve la distribución de los speed ups, comparando la ejecución en 1 pc contra la ejecución distribuida en 16 pc. Del mismo se puede apreciar que a medida que se aumenta la cantidad de bugs aumenta la mediana de los speed up. Sin embargo también incrementa la variabilidad de los speed ups cuando aumenta la cantidad de bugs.

Otro dato importante que se puede obtener del gráfico 9 es que el 75 % de los speed ups se encuentran encima de 1.0277, 1.6719, 2.7646 y 2.6313 respectivamente para cada cantidad de bugs presentes. Dado que un valor de speed up mayor a cero implica que hay mejora, entonces esto quiere decir que hay un 75 % de probabilidad como mínimo de obtener una mejora de tiempo al correr con 16 computadoras para las cantidades de bugs evaluadas.

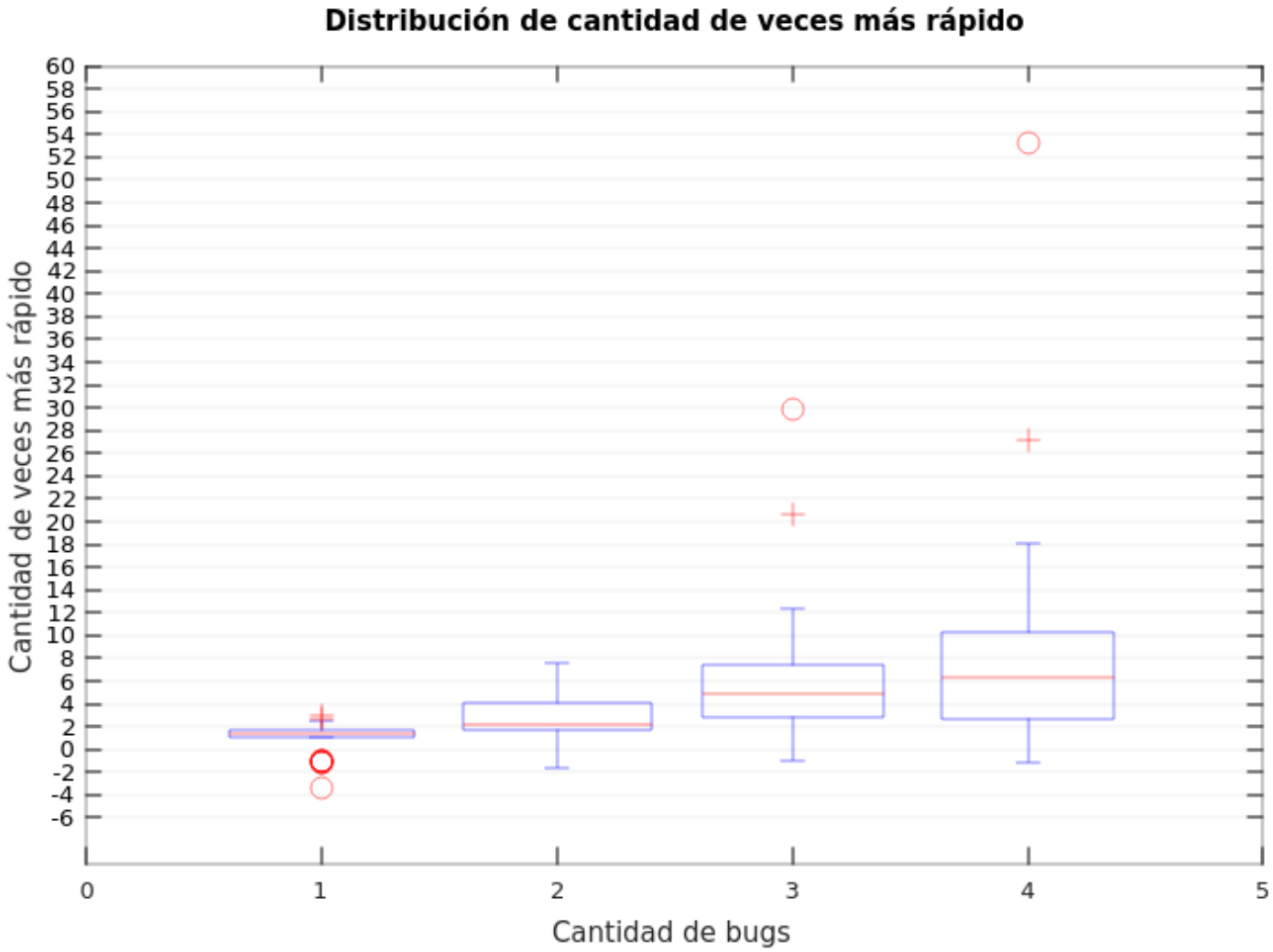


Figura 9: Distribución de speed ups según cantidad de bugs.

7.3. Optimización

La diferencia de performance entre las corridas de 16 pcs y de 32 pcs no fue tan pronunciada como se esperaba. Por esto, se hizo un análisis para ver dónde se producían los cuellos de botella. En los gráficos que se muestran en esta sección, el eje horizontal representa al tiempo desde el inicio hasta el fin de la ejecución de un test. El eje vertical puede ser el tiempo demorado en procesar un determinado input o la cantidad de elementos en las colas según se especifique en cada caso. Luego de realizar un análisis de la cantidad de elementos de las colas presentes en el servidor maestro al correr con 32 pcs se detectaron dos cuellos de botella.

7.3.1. Análisis de cuello de botella en la creación de particiones

El primer cuello de botella se encontró en la creación de particiones. En el controlador MuJavaControllerMaster se procesan los inputs que le llegan, que son de tipo “MuJavaInput”. Por cada uno de estos inputs, se genera un lote de particiones a enviar. La situación que se notó en algunos casos es que, a pesar de haber una gran cantidad de inputs “MuJavaInput”, no se incrementaba la cantidad de particiones a un ritmo suficiente para abastecer a todos los servidores esclavos libres. Un caso de esta situación se pudo identificar en el test “NodeCachingLinkedListAddFirst3Bug4Ix5Ix7D”; los estados de las colas se grafican en figura 10. A partir del segundo 78, ocurre que hay más esclavos que particiones disponibles para enviar y por lo tanto se tienen servidores esclavos que no se están aprovechando.

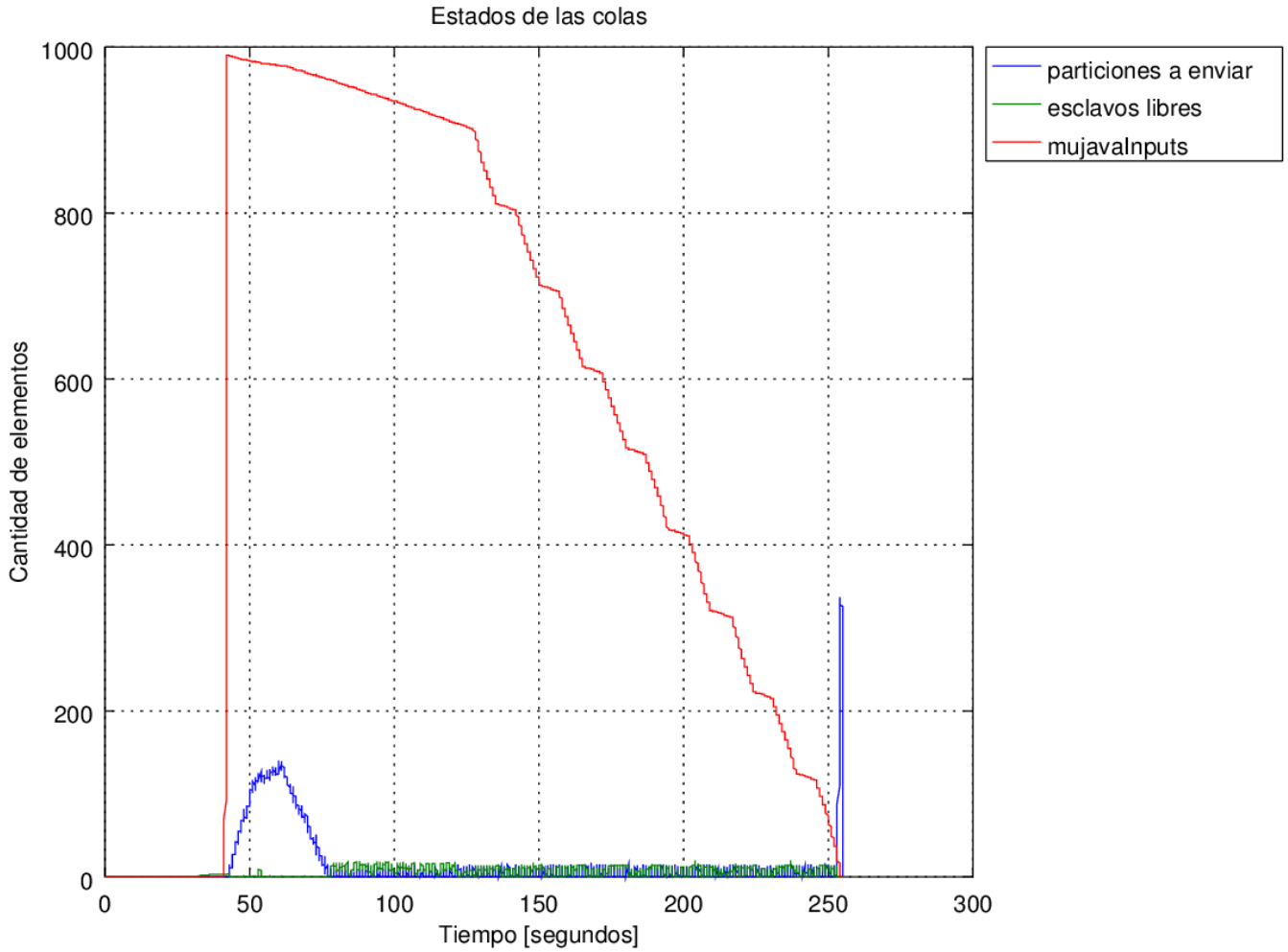


Figura 10: Cantidad de elementos en las colas del servidor maestro durante la ejecución.

Al detectar el cuello de botella en la creación de particiones, se individualizaron los tiempos de procesamiento de cada input en MuJavaControllerMaster; esto se encuentra graficado en figura 11. El tiempo de procesamiento de cada "MuJavaInput" se compone del tiempo de procesamiento de fatherize, del tiempo de creación de partición, y del tiempo de procesamiento de queueNext. De estos tres tiempos, el que produce mayor demora es el tiempo de fatherize, con tiempos de alrededor de 750 milisegundos. Esta situación se ve claramente en la figura 11 ya que los casos en los que más se demora en procesar un "MuJavaInput" coinciden con los casos en los que demora el procesamiento de fatherize.

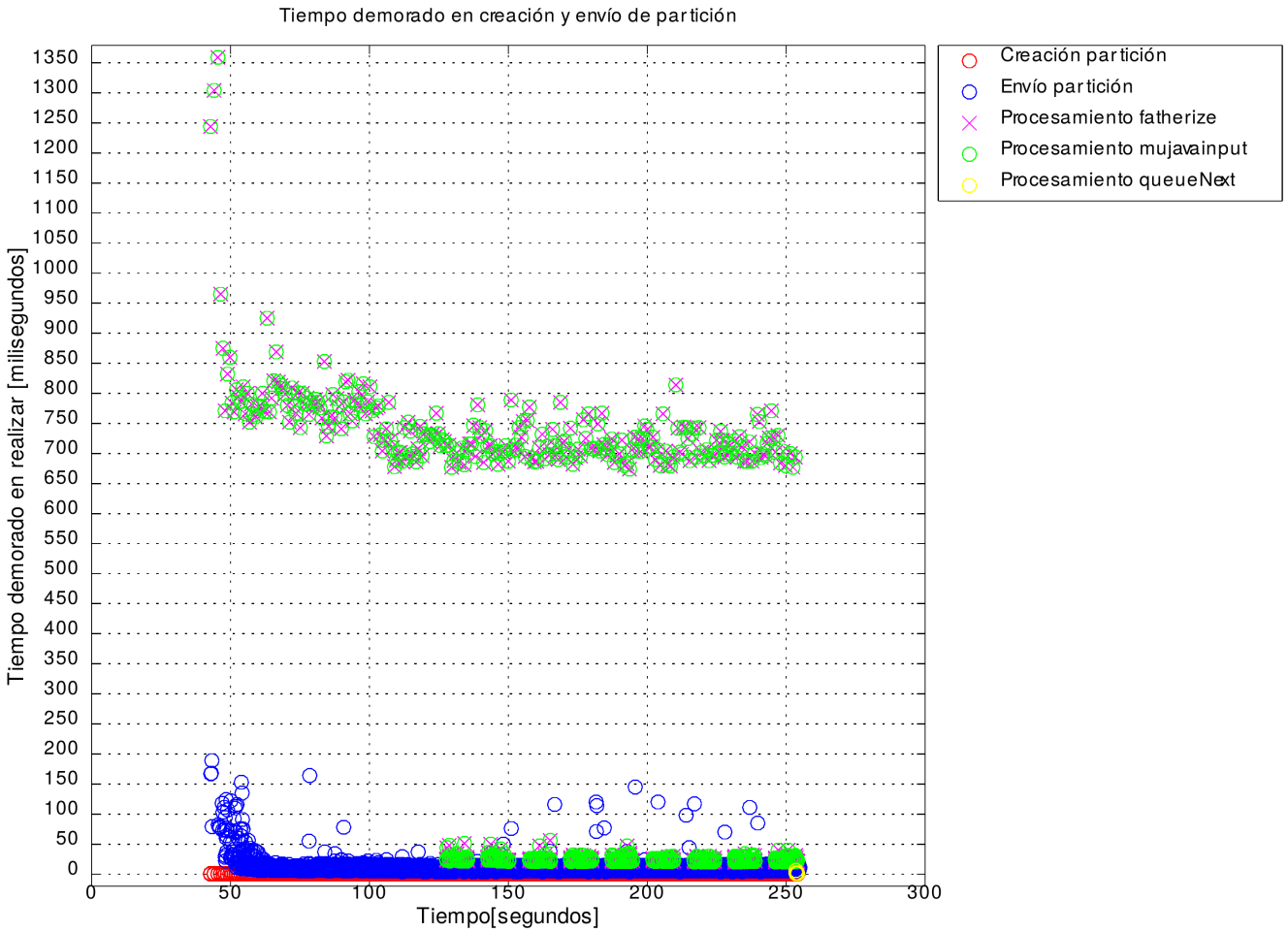


Figura 11: Tiempos de procesamiento de inputs en el servidor maestro.

Otro caso en el que sucede este cuello de botella es el test "StrykerNodeCachingLinkedListRemove3Bug19Dx30Dx36ITest". Los tiempos de procesamiento que se pueden ver en la figura 12 indican que el proceso de fatherize es el que más tiempo consume para esta prueba, con valores superiores a los 650 milisegundos en algunos casos. Debido a esta demora, se puede ver a partir de la figura 13 que entre el segundo 48 y el segundo 75 no se alcanza a crear la suficiente cantidad de particiones necesarias para aprovechar a los esclavos libres, causando que en este período se tenga entre 10 y 22 esclavos sin ocupar.

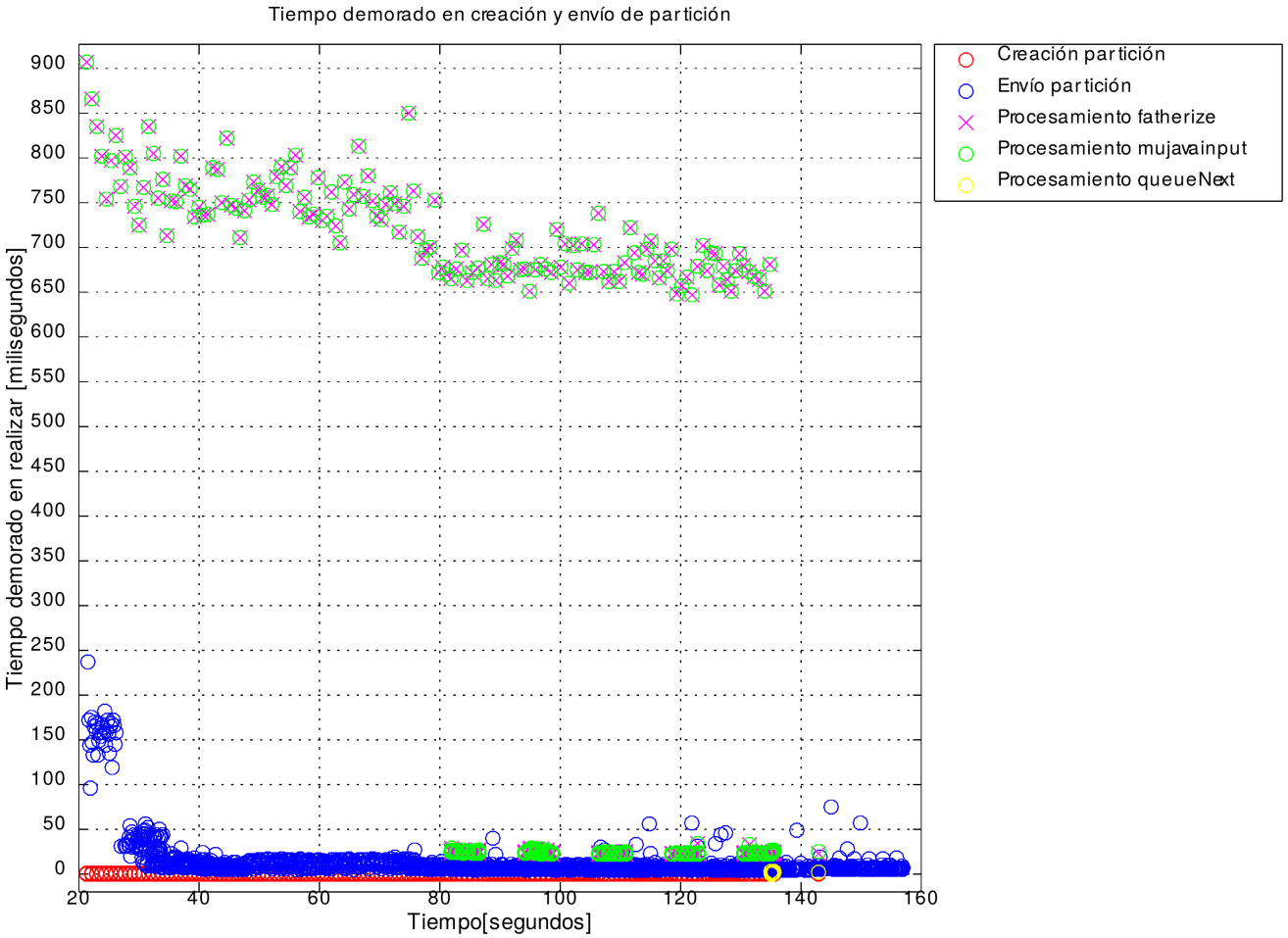


Figura 12: Tiempos de procesamiento de inputs en el servidor maestro.

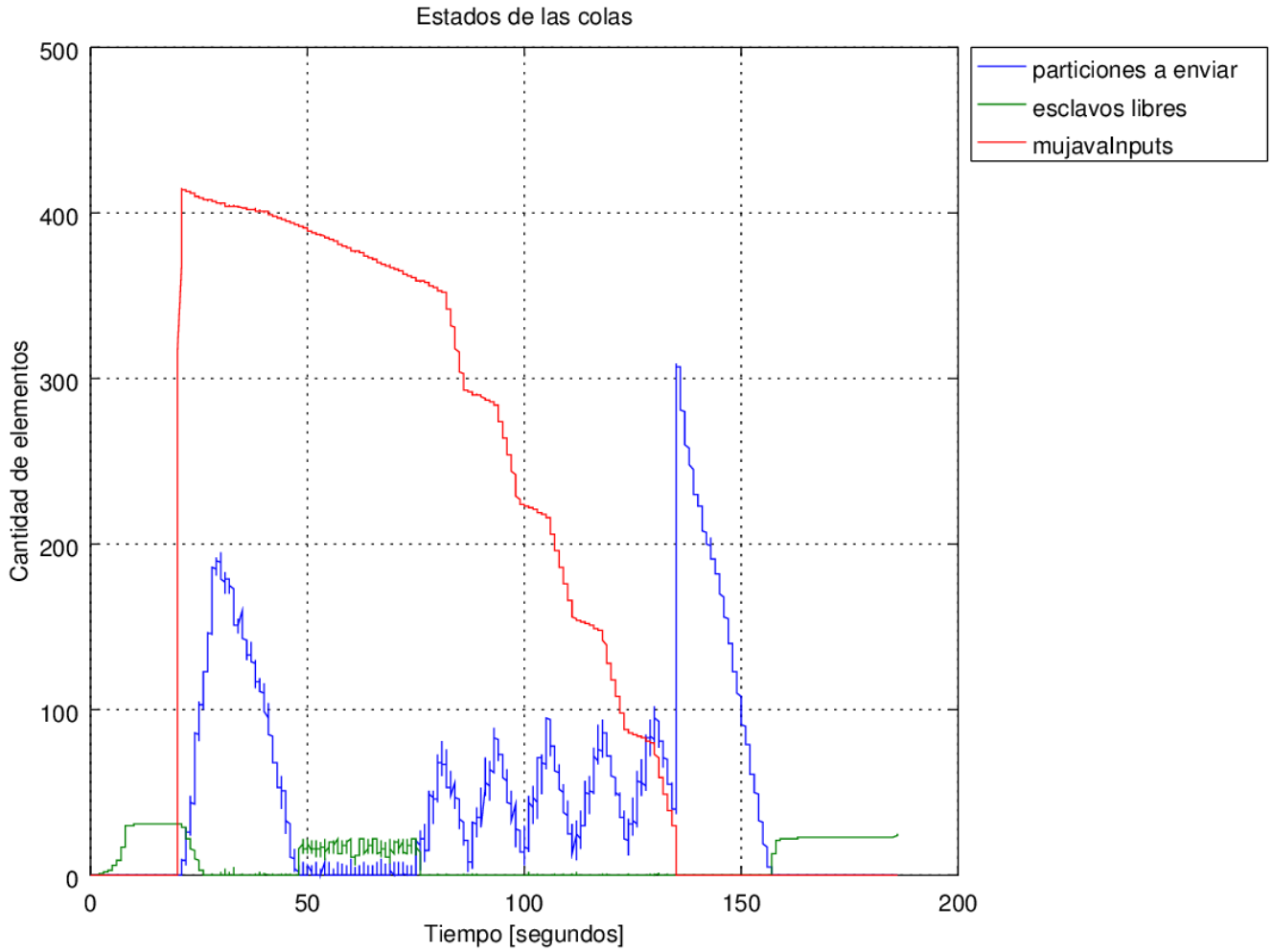


Figura 13: Cantidad de elementos en las colas del servidor maestro durante la ejecución.

7.3.2. Análisis de cuello de botella en el envío de particiones

El segundo cuello de botella se encontró en el envío de particiones. Esto se detectó en el test "StrykerBinTreeContains4Bug3Dx4Ix6Dx7DTest" al analizar la cantidad de elementos en las colas del servidor maestro a lo largo de la ejecución (ver figura 14). En este caso, se tiene una gran cantidad de particiones a enviar y a pesar de esto, se alcanzan períodos con 10 o más servidores esclavos que no se están aprovechando a partir del segundo 110.

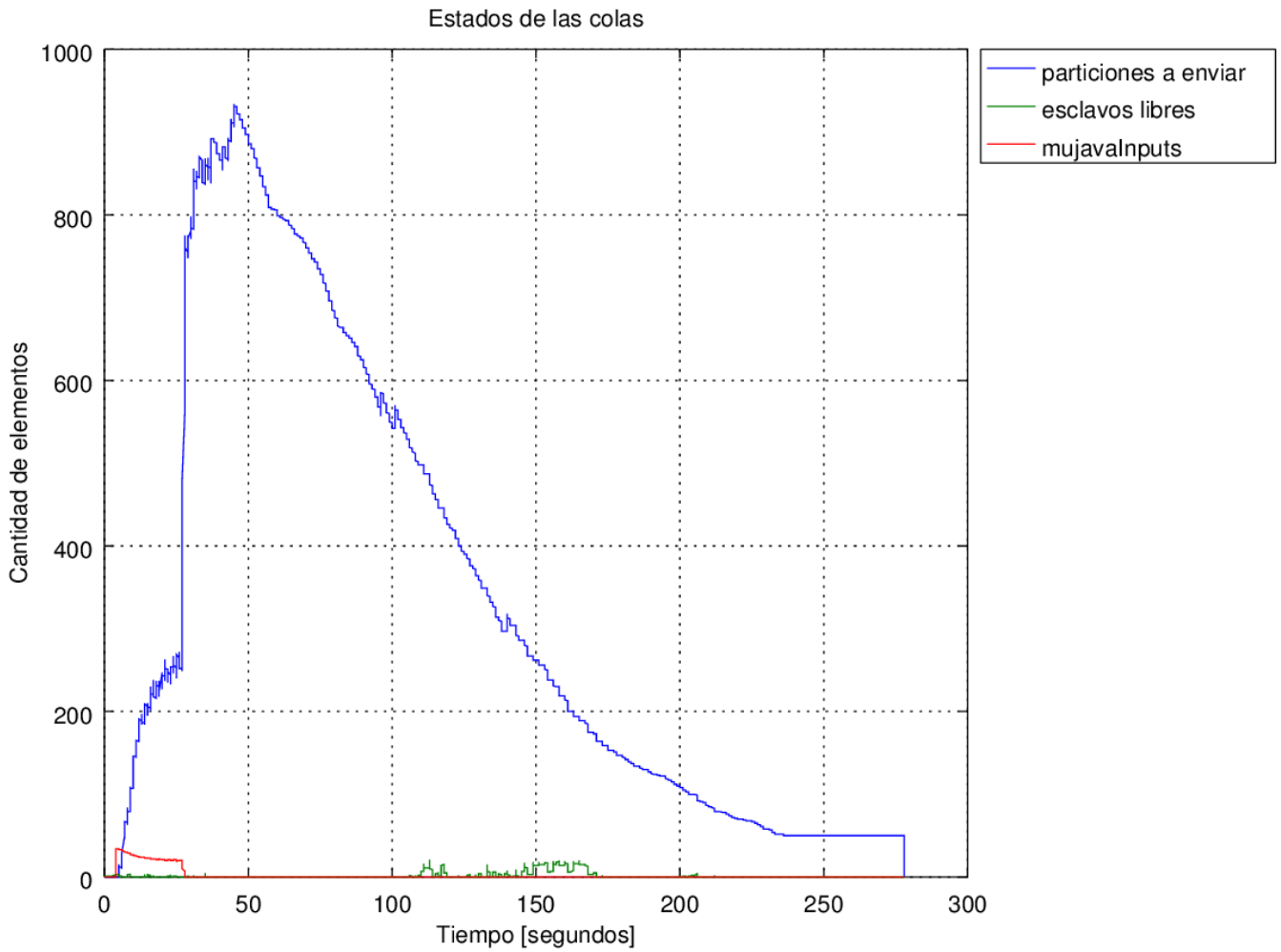


Figura 14: Cantidad de elementos en las colas del servidor maestro durante la ejecución.

Los picos de esclavos no aprovechados coinciden con las particiones que demoraron mucho tiempo en enviarse (más de un segundo), según se ve en la figura 15. El envío de cada partición se realiza en el TaskDispatcher utilizando un único hilo de ejecución; por este motivo si el envío de una partición a un esclavo demora mucho, entonces también se atrasa el envío de otras particiones a esclavos. De esta manera se genera un cuello de botella.

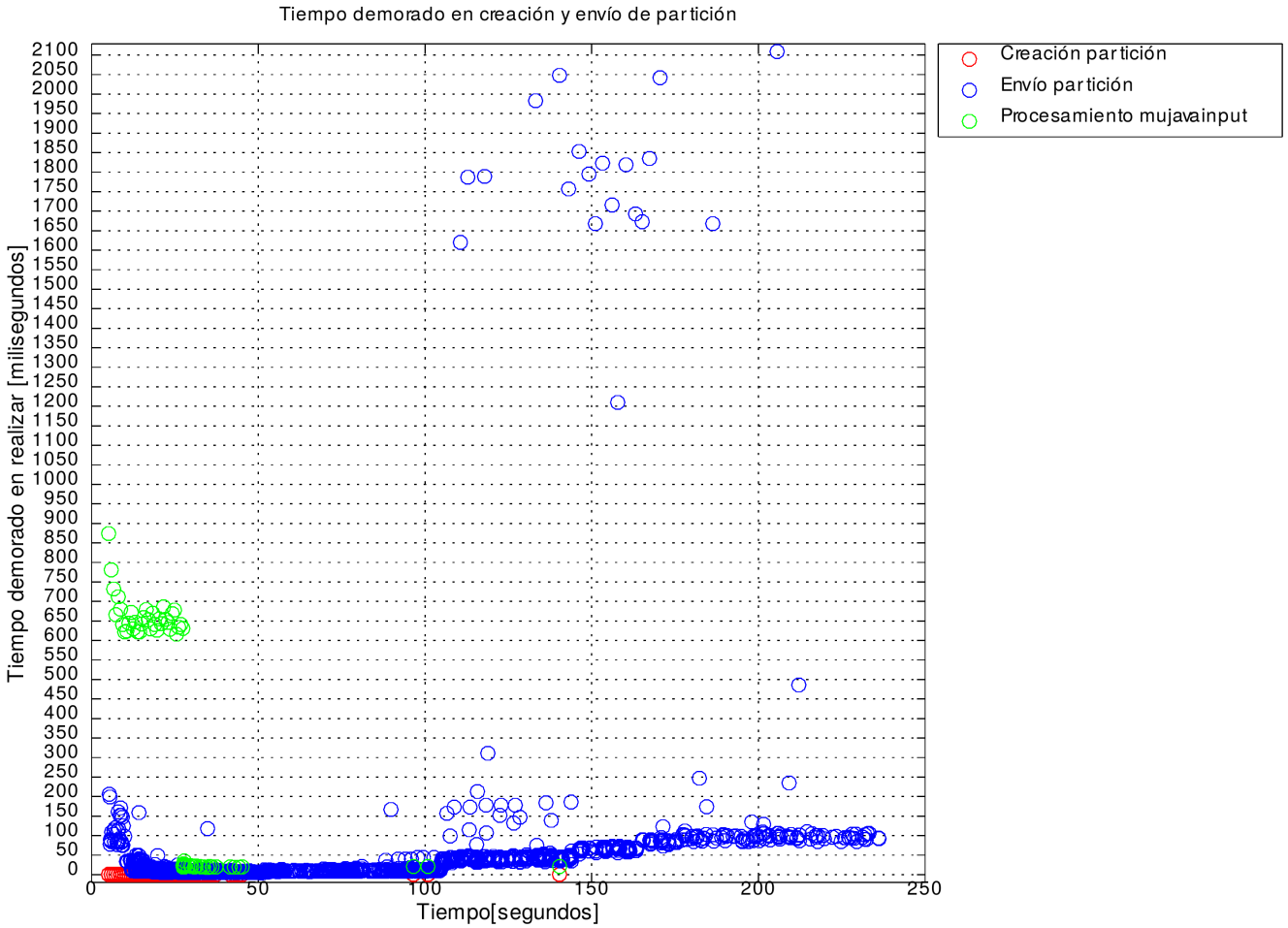
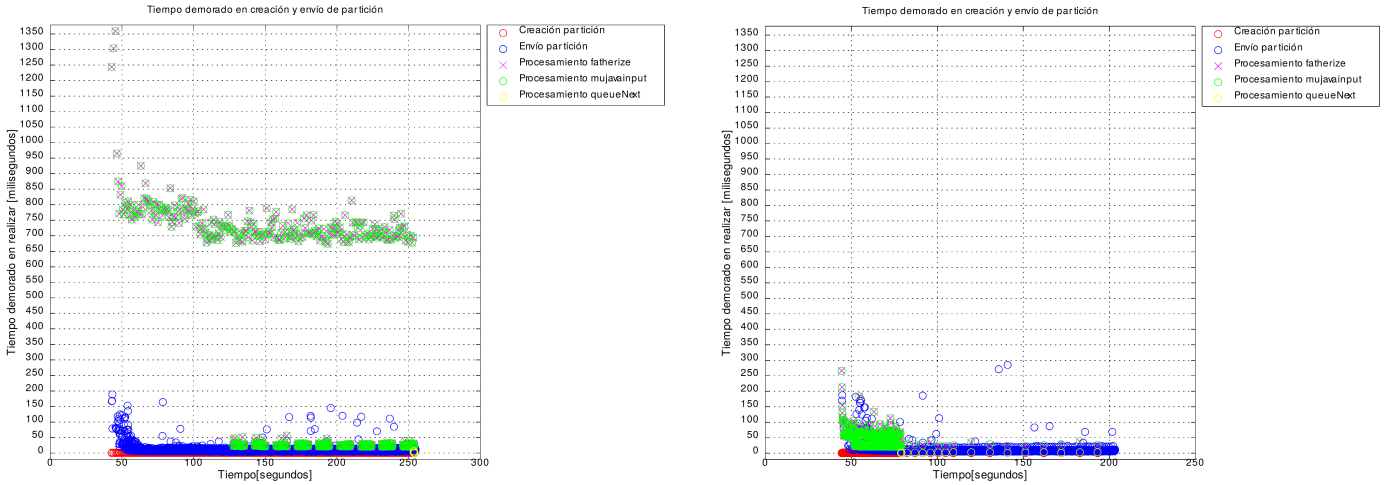


Figura 15: Tiempo demorado en el envío de cada partición.

7.3.3. Solución del cuello de botella de la creación de particiones

Para solucionar este cuello de botella se probó una posible optimización que consiste en evitar que en MuJavaControllerMaster se realice la compilación para detectar si un mutante padre no tiene descendientes que compilen antes de crear las particiones. Luego de esto cada servidor esclavo se encarga de realizar la compilación de los mutantes de la partición que recibe, descartando aquellas particiones que no tienen mutantes que compilen.

Al volver a correr el test "NodeCachingLinkedListAddFirst3Bug4Ix5Ix7D" pero ahora con la mejora, se puede notar como cae el tiempo necesario para procesar los "MuJavaInputs": antes se alcanzaban picos de 1300 milisegundos, y con la mejora los tiempos no superan los 300 milisegundos. Los tiempos con la mejora se pueden observar en figura 16b.

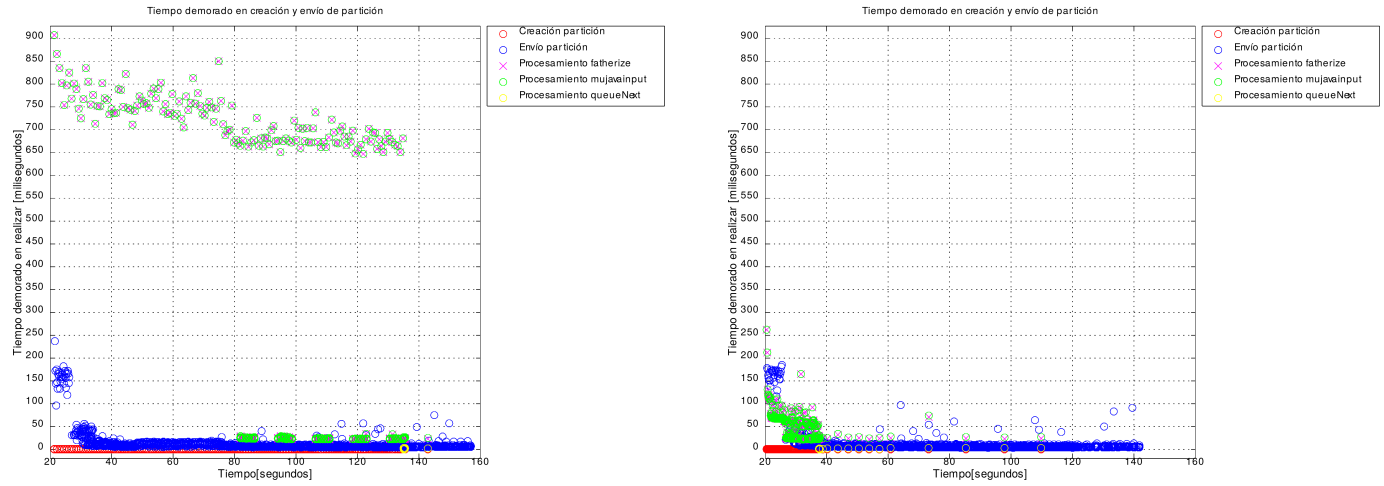


(a) Antes de la mejora.

(b) Luego de la mejora.

Figura 16: Tiempos de procesamiento comparados.

Por otra parte, al probar el código mejorado con el test "StrykerNodeCachingLinkedListRemove3Bug19Dx30Dx36ITest" se detectó que el tiempo máximo de procesamiento de "MuJavaInputs" disminuyó de 907 milisegundos a 262 milisegundos como se muestra en la figura 17.



(a) Antes de la mejora.

(b) Luego de la mejora.

Figura 17: Tiempos de procesamiento comparados.

Continuando con la evaluación de la optimización realizada, se volvieron a correr las pruebas en las que se obtuvo peor "speed up" en 16 computadoras en comparación con la corrida en una sola pc (sección 7.2.1). Como se puede apreciar en el cuadro 3, la optimización hizo que bajaran los tiempos en 16 pcs. De esta manera se pasó de tener 8 casos en los que tardaba más la versión distribuida a sólo 2. En el tercero y en el cuarto caso incluso se obtuvieron mejoras significativas en 16pcs al alcanzarse 7.35 y 12.73 de "speed up" respectivamente.

Test	Tiempo 1 pc [ms]	Tiempo 16 pc [ms]	SpeedUp 16	Tiempo 32 pc [ms]	SpeedUp 32
icse.binomialheap.set5.StrykerBinomialHeapExtractMin2Bug12Dx26DTest	362857	370973	-0.0223669379	375925	-0.0360141874
icse.binomialheap.set2.StrykerBinomialHeapExtractMin2Bug17Ix26DTest	664118	407727	0.628830075	430050	0.5442808976
icse.bintree.set2.StrykerBinTreeRemove4Bug6Dx18Dx32Dx44DTest	449172	53754	7.3560665253	44724	9.0431982828
icse.bintree.set5.StrykerBinTreeRemove3Bug21Ix26Dx40ITest	851023	61939	12.7396955069	76270	10.1580306805
icse.nodecachinglinkedlist.set1.StrykerNodeCachingLinkedListAddFirst1Bug4DTest	11890	8312	0.4304619827	8354	0.4232702897
icse.nodecachinglinkedlist.set2.StrykerNodeCachingLinkedListAddFirst2Bug4Ix6ITest	7236	6541	0.1062528665	6726	0.0758251561
icse.singlylinkedlist.set1.StrykerSinglyLinkedListContains2Bug4Dx10DTest	13538	15238	-0.1255724627	12259	0.1043315115
icse.singlylinkedlist.set4.StrykerSinglyLinkedListGetNode1Bug9ITest	5269	4666	0.1292327475	4718	0.1167867741

Cuadro 3: Tabla de resultados para peores casos 16pcs vs 1pc luego de la optimización.

Para evaluar con más detalle esta optimización se seleccionaron los casos de prueba en los que se obtuvieron valores

de speedup más bajos al comparar los resultados en 32 pcs respecto de 16 pcs. Para calcular estos valores de speedup, en la fórmula 2 se utilizó como $Tiempo_{monolítica}$ al tiempo en 16 pcs y como $Tiempo_{distribuida}$ se reemplazó por el tiempo obtenido en 32 pcs. Se tomaron los 5 casos de 3 bugs y los 5 casos de 4 bugs que tenían menor valor de speedUp.

En el cuadro 4 se pueden apreciar los resultados de estos casos antes de aplicar la optimización y en el cuadro 5 se muestran los resultados de las mismas pruebas luego de la mejora.

Test	**Bugs**	**Tiempo 1 pc [ms]**	**Tiempo 16 pc [ms]**	**Tiempo 32 pc [ms]**	**SpeedUp 16vs32**
icse.singlylinkedlist.set2.StrykerSinglyLinkedListGetNode3Bug3Dx4Dx11DTest	3	36911	18931	28733	-0.518
icse.singlylinkedlist.set2.StrykerSinglyLinkedListContains3Bug7Dx14Dx20DTest	3	43304	23909	34527	-0.444
icse.singlylinkedlist.set5.StrykerSinglyLinkedListContains3Bug5Dx7Dx11DTest	3	87303	24454	34528	-0.412
icse.nodecachinglinkedlist.set5.StrykerNodeCachingLinkedListContains3Bug2Dx5Dx7DTest	3	129658	21741	30531	-0.404
icse.singlylinkedlist.set1.StrykerSinglyLinkedListGetNode3Bug4Dx9Dx11DTest	3	164593	14913	20486	-0.374
icse.nodecachinglinkedlist.set5.StrykerNodeCachingLinkedListContains4Bug1Dx3Dx4Ix7DTest	4	227983	36232	73058	-1.016
icse.bintree.set3.StrykerBinTreeRemove4Bug4Dx14Dx31Dx43DTest	4	3224933	178906	332683	-0.860
icse.nodecachinglinkedlist.set5.StrykerNodeCachingLinkedListRemove4Bug15Dx21Dx23Dx34DTest	4	144141	55528	101687	-0.831
icse.bintree.set1.StrykerBinTreeInsert4Bug2Dx4Dx16Dx18DTest	4	1809525	217290	372367	-0.714
icse.binomialheap.set2.StrykerBinomialHeapExtractMin4Bug7Dx13Ix23Dx38DTest	4	311695	812438	1368800	-0.685

Cuadro 4: Tabla de resultados para peores casos 32pcs vs 16pcs antes de la optimización.

Test	**Bugs**	**Tiempo 1 pc [ms]**	**Tiempo 16 pc [ms]**	**Tiempo 32 pc [ms]**	**SpeedUp 16vs32**
icse.singlylinkedlist.set2.StrykerSinglyLinkedListGetNode3Bug3Dx4Dx11DTest	3	39441	18164	25145	-0.384
icse.singlylinkedlist.set2.StrykerSinglyLinkedListContains3Bug7Dx14Dx20DTest	3	43169	22492	28793	-0.280
icse.singlylinkedlist.set5.StrykerSinglyLinkedListContains3Bug5Dx7Dx11DTest	3	86896	23844	28354	-0.189
icse.nodecachinglinkedlist.set5.StrykerNodeCachingLinkedListContains3Bug2Dx5Dx7DTest	3	156033	20982	29399	-0.401
icse.singlylinkedlist.set1.StrykerSinglyLinkedListGetNode3Bug4Dx9Dx11DTest	3	163448	14041	15712	-0.119
icse.nodecachinglinkedlist.set5.StrykerNodeCachingLinkedListContains4Bug1Dx3Dx4Ix7DTest	4	226925	32885	27887	0.179
icse.bintree.set3.StrykerBinTreeRemove4Bug4Dx14Dx31Dx43DTest	4	74477	71482	61656	0.159
icse.nodecachinglinkedlist.set5.StrykerNodeCachingLinkedListRemove4Bug15Dx21Dx23Dx34DTest	4	139548	52916	48522	0.091
icse.bintree.set1.StrykerBinTreeInsert4Bug2Dx4Dx16Dx18DTest	4	1622278	201275	193516	0.040
icse.binomialheap.set2.StrykerBinomialHeapExtractMin4Bug7Dx13Ix23Dx38DTest	4	343591	796594	397913	1.002

Cuadro 5: Tabla de resultados para peores casos 32pcs vs 16pcs luego de la optimización.

En la figura 18 se puede apreciar la comparación de tiempos de ejecución promedio antes y después de la mejora en estos casos seleccionados. Se representó con el mismo color las curvas para la misma cantidad de pcs pero con una línea punteada se graficó los valores anteriores a la mejora.

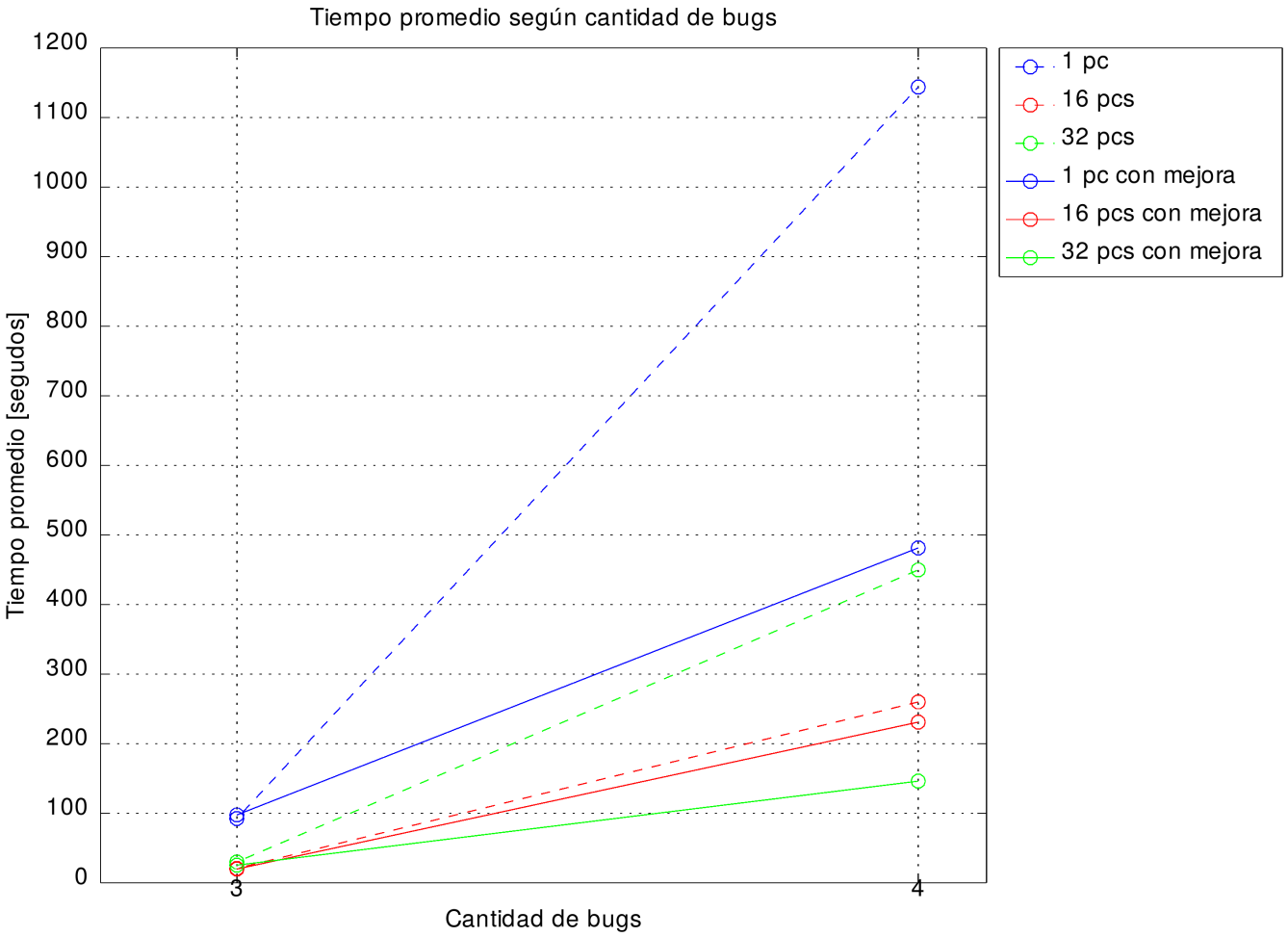


Figura 18: Promedio de tiempos según la cantidad de pc comparando antes y después de la optimización.

En general para estos casos mejoraron los tiempos promedios para las tres cantidades de computadoras (1,16 y 32 computadoras). Además, antes de la mejora los tiempos para 32 pcs en estos casos eran mayores que utilizando 16 pcs, pero ahora con la mejora se revirtió esto en los casos de 4 bugs, obteniendo menores tiempos con 32 computadoras.

Con esta optimización se espera que los tiempos promedio sean todavía menores en 32 pcs que en 16 pcs al promediar todos los tests.

8. Conclusión

En este documento se presentó una mejora a la herramienta Stryker que permite realizar escalamiento horizontal. También se presentó una técnica de poda distribuida que es compatible con la poda que tenía Stryker originalmente.

La poda distribuida consiste en dos partes. La primera es el salteo de mutaciones basado en cálculo de feedback (realizado en servidores esclavos) para obtener la siguiente partición (en el servidor maestro). La segunda parte es la cancelación de particiones que se estaban por enviar y las que ya habían sido enviadas.

Se evaluó la herramienta distribuida en un conjunto de clases de prueba de Stryker que implementan cuatro estructuras de datos en Java equipadas adecuadamente con contratos JML, mostrando la efectividad de la técnica para disminuir los tiempos al aumentar la cantidad de computadoras que se utilizan simultáneamente.

Durante las pruebas se determinó que aquellas que son más largas exhiben mayor beneficio al utilizar esta técnica de distribución. En términos generales, antes de aplicar la optimización se observó que resulta más conveniente utilizar 16 computadoras a partir de los 3 bugs y 32 computadoras a partir de 4 bugs en el método a analizar.

En los casos en los que se evaluó Stryker distribuido con la optimización, disminuyeron los tiempos y así se logró que la diferencia de tiempos entre utilizar 16 pcs y 32 pcs sea mayor.

9. Trabajos futuros

A la implementación presentada en este documento para la versión distribuida de Stryker se le pueden agregar mejoras para aumentar el desempeño de la herramienta y la facilidad de uso.

9.1. Mejora en la división de tareas

Entre las mejoras que apuntan al desempeño, se encuentra el agregado de mayor inteligencia al algoritmo de particionamiento para evitar que se distribuyan particiones muy pequeñas o particiones muy grandes. Cuando se distribuyen muchas particiones pequeñas, es decir particiones que incluyen pocos mutantes a explorar, los servidores esclavos consumen más tiempo comunicándose con el servidor maestro para recibir los datos de las particiones. Esto se podría solucionar distribuyendo múltiples particiones pequeñas a cada servidor esclavo en un mismo mensaje.

Por otra parte cuando las particiones son muy grandes, en algunos casos ocurre que pocos servidores esclavos analizan grandes particiones mientras que los demás no tienen tareas para analizar; esto se podría solucionar distribuyendo particiones más pequeñas para distribuir mejor la carga de trabajo entre los servidores esclavos.

9.2. Apagado automático

En la implementación actual, el clúster se apaga automáticamente cuando se detecta que se encontró una solución y se especificó que no se necesita buscar múltiples soluciones. Adicionalmente, se puede mejorar la usabilidad de la herramienta agregándole un sistema que detecte cuando se agota el espacio de búsqueda para notificar al usuario y luego apagar el clúster completo en forma automática. Implementar esto último resulta complejo ya que la herramienta Stryker en su estado original no presenta un mecanismo que detecte el agotamiento del espacio de búsqueda.

9.3. Tolerancia a falla de esclavos

También se le puede incrementar la tolerancia a fallas implementando un mecanismo que detecte si un esclavo pasó a estar fuera de línea para poder reasignar la tarea que tenía dicho esclavo a otro que se encuentre desocupado.

10. Referencias

[1] Y.-S. Ma, J. Offutt and Y.-R. Kwon. *MuJava : An Automated Class Mutation System.* Journal of Software Testing, Verification and Reliability, 15(2), Wiley, 2005.

[2] J.P. Galeotti, N. Rosner, C. López Pombo, M. Frias. *TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds.* IEEE TSE 39(9): 1283-1307 (2013).

[3] G. T. Leavens, Y .Cheon, C. Clifton, C. Ruby, D. R. Cok. *How the Design of JML Accomodates Both Runtime Assertion Checking and Formal Verification.* FMCO 2002: 262-284.

[4] L.R. Zemin, M. F. Frias, N. Aguirre, S. Brida, A. Mili, A. Jaoua, S. Perez De Rosso (2015). *Stryker: Scaling multiline specification-based program repair by prunning infeasible mutants with SAT.* ITBA, unpublished.

[5] M. E. Turrín, L. I. Gomez. *Clase 3: RMI, dynamic classloading.*Segundo cuatrimestre 2015,Programación de objetos distribuidos, ITBA. Presentación de Microsoft PowerPoint.

[6] M. F. Frias. *Stryker: a tool for automated bug fixing based on sat-solving.*Primer cuatrimestre 2016, Análisis y verificación de modelos y códigos. ITBA. Presentación de Microsoft PowerPoint.

[7] M. A. Buquete, J. M. Sotuyo Dodero. *Teoría 3.*Segundo cuatrimestre 2015, Ingeniería del software II. ITBA.Presentación de Microsoft PowerPoint.

[8] *Trail: RMI.* Oracle,The Java™ Tutorials, 2017.

[9] W. Grosso. *Java RMI.* O' Reilly, 2001.

[10] M. Fowler. *Uml Destilled.* Tercera edición. Addison Wesley, 2003.

[11] S. W. Ambler. *Agile Modeling: effective practices for modeling and documentation.* Ambysoft Inc, 2014.