JIS-JVMPI per Linux IA32: Instrumentació d'aplicacions Java en un entorn Linux

David Carrera, Jordi Guitart, Javier Bartolomé, Jordi Torres, Eduard Ayguadé

Centre Europeu de Paralelisme de Barcelona (CEPBA)
Departament d'Arquitectura de Computadors,

Universitat Politècnica de Catalunya,

C/ Jordi Girona, 1-3, Campus Nord UPC, Mòdul C6,
E-08034 Barcelona (Espanya)

{dcarrera, jguitart, jbartolo, torres, eduard}@ac.upc.es

Índex

L'entorn Java va ser desenvolupat tenint com a principals objectius la portabilitat i la facilitat d'ús. Aquests objectius, assolits en major o menor grau ja des dels seus orígens, han rebut una gran acollida pel que fa al col·lectiu de desenvolupadors d'arreu del món. Aquest fet ha propiciat que la plataforma Java s'hagi vist fortament impulsada pels seus propis usuaris i li ha atorgat un important nivell de popularitat alhora que l'ha fet arribar a nous entorns d'execució per als quals no havia estat pensada inicialment.

La computació d'altes prestacions té actualment com a suport hardware habitual les arquitectures multiprocessadores. Java, per la seva banda, té molt arrelat a la seva filosofia de treball la programació *multithread*, capaç d'aprofitar fàcilment les prestacions de les màquines (i sistemes operatius, no cal oblidar-ho) que disposen de suport per a multiprocés simètric (SMP). Tot i així, Java actualment es troba explotat principalment en màquines monoprocessadores, ja que s'e n'intenta treure més profit de la seva alta portabilitat més que les prestacions en arquitectures paral·leles.

Actualment, la plataforma Java presenta una certa carència d'eines per a l'estudi del rendiment de les aplicacions basades en ella amb un nivell de detall físic adequat. Tot i que existeixen nombroses eines per a estudiar aplicacions escrites en Java, aquestes acostumen a presentar moltes limitacions pel que fa a la informació de baix nivell que poden presentar. Això comporta que sigui possible identificar els colls d'ampolla que es formen en els fluxos d'execució d'aquestes aplicacions però complica la tasca d'identificar les causes que porten a la generació d'aquests colls d'ampolla. És necessari, doncs, l'ús d'una eina adequada per a estudiar amb detall el comportament de les aplicacions basades en Java que cal millorar, per a que la informació que se'n pugui extreure sigui la màxima possible.

En aquest article es presenta el procés de creació de JIS (Java Instrumentation Suite) per a Linux, una eina d'instrumentació d'aplicacions Java que pretén venir a cobrir els aspectes que la resta d'eines similars existents en el mercat obliden i que s'orienta cap a l'estudi de Servidors Web i Servidors d'Aplicacions escrits en Java. S'expliquen les decisions preses per al seu desenvolupament i se'n presenten els detalls tècnics més destacables.

1 Introducció

L'entorn Java va ser desenvolupat tenint com a principals objectius la portabilitat i la facilitat d'ús. Aquests objectius, assolits en major o menor grau ja des dels seus orígens, han rebut una gran acollida pel que fa al col·lectiu de desenvolupadors d'arreu del món. Aquest fet ha propiciat que la plataforma Java s'hagi vist fortament impulsada pels seus propis usuaris i li ha atorgat un important nivell de popularitat alhora que l'ha fet arribar a nous entorns d'execució per als quals no havia estat pensada inicialment.

La portabilitat entre plataformes ha estat durant molt de temps un objectiu no del tot contemplat en els desenvolupaments quan la prioritat era la facilitat de desenvolupament i l'increment de rendiment de les aplicacions produïdes. Un cop aquests altres objectius havien estat assolits en un grau acceptable, el tema de la portabilitat va esdevenir una nova àrea d'interès. De seguida va ser vist que la portabilitat requeria la generació de nous nivells d'abstracció en l'habitual arquitectura del software que es venia desenvolupant ja que en algun punt s'havia d'aïllar el codi de l'aplicació de les característiques particulars de cada màquina sobre la qual havia de córrer.

Java va venir a omplir, per primer cop de manera massiva, el forat que hi havia en el món del desenvolupament de software pel que fa a temes de portabilitat. Va néixer com un entorn multiplataforma, amb especificacions lliures per a la comunitat amb l'objectiu de no frenar-ne la ràpida a expansió i amb l'abstracció necessària que possibilitava el desenvolupament d'aplicacions totalment independents de l'entorn en el qual s'executessin: la Màquina Virtual de Java (Java Virtual Machine, JVM).

Aquesta nova abstracció en l'arquitectura d'execució de les aplicacions, propicia també l'obertura d'un nou camp d'investigació pel que fa a les ciències de la computació: la millora del rendiment de l'esmentada capa d'abstracció per a permetre el màxim aprofitament de la plataforma hardware sobre la que aquesta es troba corrent.

La computació d'altes prestacions té actualment com a suport hardware habitual les arquitectures multiprocessadores. Java, per la seva banda, té molt arrelat a la seva filosofia de treball la programació *multithread*, capaç d'aprofitar fàcilment les prestacions de les màquines (i sistemes operatius, no cal oblidar-ho) que disposen de suport per a multiprocés simètric (SMP). Tot i així, Java actualment es troba explotat principalment en màquines monoprocessadores, ja que s'utilitza amb l'objectiu d'explotar-ne l'alta portabilitat més que les prestacions en arquitectures paral·leles.

La plataforma Java presenta una certa carència d'eines per a l'estudi del rendiment de les aplicacions basades en ella amb un nivell de detall físic adequat. Tot i que existeixen nombroses eines per a estudiar aplicacions escrites en Java, aquestes acostumen a presentar moltes limitacions pel que fa a la informació de baix nivell que poden presentar. Això comporta que sigui possible identificar els colls d'ampolla que es formen en els fluxos d'execució d'aquestes aplicacions però complica la tasca d'identificar les causes que porten a la generació d'aquests colls d'ampolla. **És necessari, doncs, l'ús d'una eina adequada per a estudiar amb detall el comportament de les aplicacions basades en Java que cal millorar, per a que la informació que se'n pugui extreure sigui la màxima possible.**

Ens trobem, doncs, en un entorn en què la plataforma Java ha triomfat àmpliament en molts sectors de la informàtica però encara es troba amb barreres d'entrada en certs àmbits que presenten alts requeriments computacionals. Aquesta limitació ve donada en gran mesura per la falta d'experiència i de casos coneguts d'explotació de Java en entorns amb majors requisits que els habituals. D'altra banda, les arquitectures que habitualment donen suport a aquests entorns d'execució tampoc es troben àmpliament estudiats per a treure'n tot el profit possible des de la màquina virtual de Java. Demostrar que aquests entorns també són abastables, encara que considerant certes limitacions, per Java i trobar eines flexibles que permetin entendre el comportament de les aplicacions Java en aquestes plataformes són, tal com es presenta amb més detall a continuació, els objectius d'aquest document.

1.1 Barcelona Java Suite

El Barcelona Java Suite (BJS) és una projecte de recerca en polítiques i mecanismes de planificació de cara a buscar l'execució eficient d'aplicacions Java *multithread* corrent sobre servidors paral·lels o distribuïts, en els quals la interacció i coordinació entre diferents nivells (aplicació, compilador, llibreria de *threads* i kernel) han de ser estudiats minuciosament. El BJS fa ús de les eines prèviament desenvolupades pel CEPBA. L'entorn integra tècniques i eines proposades en diferents entorns de recerca, possibilitant l'explotació del seu potencial conjunt per al desenvolupament de noves idees i arquitectures.

Les línies de desenvolupament passades del BJS cobrien, principalment, les extensions de la plataforma Java per a incrementar-ne l'explotació d'entorns paral·lels, la cooperació entre la màquina virtual de Java i el sistema operatiu i la creació d'un entorn d'avaluació i anàlisi del rendiment.

Actualment, les línies de recerca del grup s'ha reorientat lleugerament, ampliant-ne els objectius i enfocant-se cap a l'estudi dels entorns basats en servidors d'aplicacions, com ara Tomcat i WebSphere. També es contemplen d'altres noves tecnologies, encara exploratòries i emergents, com per exemple, els WebServices.

1.2 Presentació de l'entorn de treball

1.2.1 JIS-JVMPI

L'estudi d'aplicacions escrites en Java pot ser enfocat des de diferents punts de vista. Un d'ells és el del rendiment. Un altre el del comportament. Aquests dos punts de vista intenten ser coberts per una eina desenvolupada al llarg d'aquest projecte, anomenada JIS-JVMPI, que pretén afegir llum a un punt fosc com és el comportament de les aplicacions Java un cop es troben en estat d'execució sobre la Màquina virtual de Java.

El desenvolupament JIS-JVMPI va ser iniciat temps enrera en el sí del CEPBA[6] com a PFC d'un estudiant de la Facultat d'Informàtica de Barcelona[?]. Actualment, ha estat reescrit de zero per a donar cabuda a noves necessitats en la instrumentació d'aplicacions Java.

Aquesta eina pretén servir per a instrumentar aplicacions escrites en Java. Aquesta instrumentació es fonamenta en l'ús d'una eina de visualització de traces creada pel CEPBA que s'anomena Paraver[8], presentada amb més detall en el punt 1.2.2.

Què és el JVMPI?

El *Java Virtual Machine Profiler Interface*[9] és una facilitat que ofereix la màquina virtual de Java de cara a extreure informació d'una execució concreta.

Consisteix en un enllaç que s'estableix en temps de càrrega de la màquina virtual entre aquesta i una llibreria compartida generada per l'usuari que vol instrumentar l'execució a través del qual es notifica a l'esmentada llibreria tots aquells events que es produeixen dins de la màquina virtual. Un esquema de funcionament es pot veure a la figura 1.1.

La llibreria d'instrumentació pot decidir dinàmicament quins events vol que li siguin notificats i quins no. En funció d'aquestes decisions, el cost de la instrumentació serà major o menor.

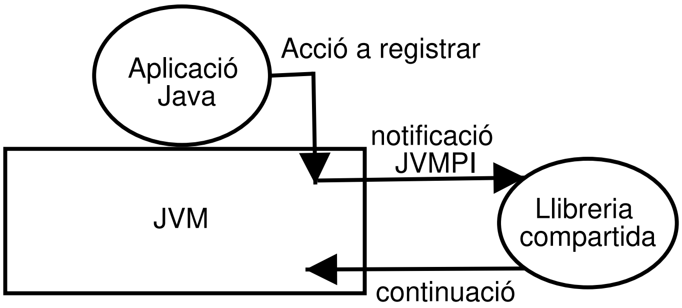

Figura 1.1: Esquema de funcionament del JVMPI

El codi generat per l'usuari que instrumenta és el responsable de decidir què es fa amb la informació que li és reportada pel JVMPI. D'aquesta manera, la instrumentació realitzada és transparent per a l'execució de codi dins de la màquina virtual de Java, ja que només té constància d'una interfície comú acordada amb la llibreria i que es troba clarament especificada per Sun.

Què és JIS?

JIS significa Java Instrumentation Suite. Aquesta eina d'instrumentació, desenvolupada dins del grup BJS[7] (Barcelona Java Suite) del CEPBA, té com a plataforma objectiu Java. Es troba desenvolupada sobre diferents sistemes (SGI - Irix, IBM-Aix i Linux), de manera que es poden estudiar diferents entorns d'execució per a una mateixa aplicació Java, i es troba pensada concretament per a l'estudi d'aplicacions paral·leles.

Existeixen dos branques principals de desenvolupament de JIS. Per una banda es troba la versió original desenvolupada per a SGI, la qual es fonamentava en el concepte d'interposició dinàmica de codi[2] (basada les DiTools[4] del CEPBA) per a obtenir informació sobre l'execució de l'aplicació Java que es pretén estudiar. Per altra banda, existeixen unes altres versions desenvolupades a partir del JVMPI (veure 1.2.1 a la pàgina anterior *"Què és el JVMPI"*), menys dependents de la plataforma i per tant més portables.

Què és JIS-JVMPI per Linux IA32?

La versió de JIS-JVMPI per Linux IA32 (*Intel Architecture 32 bit*) és la que s'ha desenvolupat al llarg d'aquest projecte. Aquesta versió parteix d'un anterior projecte de final de carrera desenvolupat al CEPBA i que ja utilitzava el JVMPI per a instrumentar aplicacions Java sobre Linux.

La versió anterior de JIS-JVMPI sobre Linux era plenament funcional però presentava certes carències derivades del funcionament del JVMPI que n'impedien l'ús de cara a realitzar estudis amb profunditat d'aplicacions Java. Aquestes limitacions venien donades pel fet que la instrumentació que permet realitzar el JVMPI es basa sempre en informació extreta a partir de les estructures internes de la màquina virtual de Java que, finalment, no és més que un procés que corre en espai d'usuari. Per això no permet notificar certes informacions de sistema (informacions sobre l'estat de les estructures del nucli del sistema operatiu) molt valuoses de cara a estudiar el rendiment de les aplicacions Java.

Per a aconseguir extreure informació del sistema, la nova versió implementada de JIS-JVMPI té una estructura mixta: combina dades notificades per la màquina virtual a través del JVMPI amb dades extretes del nucli del sistema a través d'un sistema de col·laboració entre processos de sistema i d'usuari que més endavant s'explicarà, i que treu partit del fet que Linux es distribueix sota llicència GNU[10], de manera que és de codi obert, sobre el qual es poden realitzar modificacions lliurement.

Amb aquesta combinació, s'aconsegueixen fusionar informacions de molt baix nivell amb aquelles d'alt nivell extretes de la màquina virtual. Això permet dotar d'un major significat les dades de baix nivell i associar-les a events propis de les aplicacions Java.

La implementació d'aquesta nova versió de JIS-JVMPI, iniciada com una mera ampliació, ha suposat un un redisseny i reescriptura total de l'eina per tal d'assolir els nous objectius marcats. L'arquitectura ha estat modificada per a donar cabuda a la integració de dades entre dos nivells, usuari i sistema, i fer-ne la fusió en temps d'execució.

1.2.2 Paraver

Paraver[8] és una eina de visualització i anàlisis de traces desenvolupada en el CEPBA. Tant el programa com el format de les seves traces es troben orientades a l'estudi de programes *multithread* executant-se en entorns paral·lels, tot i que altres usos poden ser-li donats.

L'element principal que representa l'eina és la traça. Una traça no és més que una seqüència de *records* ordenats temporalment i que representen informació sobre events succeïts en el sistema al llarg d'una execució d'un programa. Les traces de Paraver es composen principalment de tres tipus d'elements:

- Estats: Indiquen estats de *threads* del sistema. Cada estat porta associat, entre altres coses, un identificador de *thread*, un identificador de processador, un temps d'inici, un temps de final i l'estat que s'ha de representar.

- Events: Un event Paraver representa un event puntual que ha tingut lloc en el sistema durant l'execució. Cada event porta associat un identificador de *thread*, un identificador de processador, un temps d'event, un identificador de tipus d'event i un valor d'event.

- Comunicacions: Un event de comunicació representa, de manera molt general, una relació entre dos *threads*. A nivell pràctic, van ser introduïts per a representar un pas de missatges entre *threads*. L'aplicació que s'ha donat a aquest tipus de *records* en aquest projecte ha estat lleugerament diferent, tal com es veurà al llarg d'aquest document. Cada event de comunicació consta, tant per a l'origen de la comunicació com per a la destinació, de temps d'event, *thread*, processador, identificador de missatge i mida del missatge.

A partir d'aquestes dades, Paraver representa la traça situant els *threads* en l'eix vertical i els events i estats en l'eix horitzontal, tal com es pot veure en la figura 1.2. En ella, el color vermell representa que un *thread* es troba en estat BLOCKED, color blau clar indica que el *thread* es troba en estat de READY, color blau fosc indica que el *thread* es troba en estat de RUN i color negre indica que no hi ha cap *thread* representat en aquell moment en aquella posició. Els events es representen com a petites banderetes en la banda superior de l'estat de cada *thread* i els events de comunicació es presenten coma línies longitudinals grogues que van d'un *thread* a un altre.

També es pot obtenir informació numèrica sobre els events i estats fent-hi *click* al damunt amb el ratolí. D'altra banda, Paraver permet realitzar moltes modificacions (i combinacions d'elles) sobre les dades representades. Això permet, per exemple, fer que l'estat d'un *thread* passi a ser el valor de l'últim event aparegut per al *thread* enlloc de l'estat definit en la traça, així com filtrar events, estats, *threads*... de manera que l'estudi que possibilita sobre les traces obtingudes és molt ampli.

D'altres eines es troben dins de Paraver, com ara analitzadors estadístics de les traces, tant sobre una finestra com realitzant correlacions entre dades de dues finestres diferents. Així, l'estudi estadístic que es pot realitzar manualment sobre la traça pot ser obtingut, si no directa, gairebé directament des de dins de la pròpia eina.

Un detall a destacar aquí, és el **reús que fa JIS dels slots de *threads*** de què disposa Paraver: un cop un *thread* ha finalitzat, l'*slot* que ocupava es torna a fer servir per a posteriors *threads* que entrin en el sistema. Aquest fet es pot veure en la figura 1.2, on el cicle de vida dels *threads* etiquetats

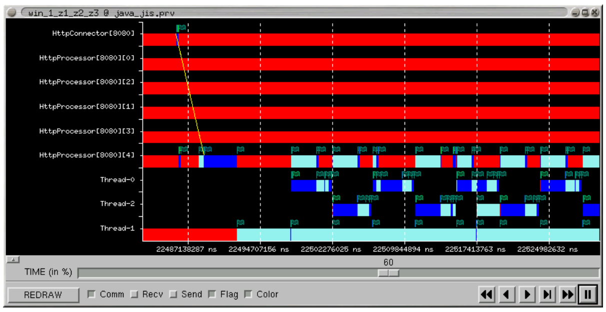

Figura 1.2: Exemple representació d'una de traça Paraver

com *Thread-0, Thread-1* i *Thread-2*, inicia i finalitza diverses vegades. Això indica, que són nous *threads* que han entrat en el sistema.

El que resta de text es troba estructurat tal com segueix:

- Apartat 2: En aquest capítol s'introdueixen els antecedents més directes dels dos objectius principals que cobreix aquest projecte, i analitza quins són els requeriments bàsics que cal imposar-se.

- Apartat 3: Es cobreix el desenvolupament tècnic del projecte. Es presenten les diferents decisions de disseny preses i els detalls bàsics de la implementació realitzada. Aquest capítol es troba dividit en dues parts, cadascuna corresponent a un dels dos objectius principals marcats per al desenvolupament del projecte.

- Apartat 4: Aquest capítol abraça, d'una banda, els resultats obtinguts a partir de les modificacions realitzades sobre el servidor Web Tomcat i, d'una altra banda, les aplicacions donades a l'eina desenvolupada (JIS) per a la correcta instrumentació d'aplicacions Java.

- Apartat 5: Presenta les conclusions sobre el treball realitzat i futures línies de desenvolupament per a JIS.

2 Anàlisi d'antecedents i de requeriments

2.1 Creació d'eines d'instrumentació d'aplicacions Java

2.1.1 Antecedents

En aquest punt es presenten els antecedents més directes d'eines d'instrumentació de codi Java semblants a JIS.

JIS per a SGI

JIS (Java Instrumentation suite) no és una eina nova, conceptualment parlant, dins del BJS[7]. Rera el nom JIS existeix principalment una filosofia d'instrumentació, independentment de la plataforma i implementació concreta de què es parli.

La primera versió de JIS desenvolupada va ser la basada en la plataforma SGI (Silicon Graphics Inc.) corrent Irix (el sistema UNIX de SGI). Aquella versió de JIS, de la qual es pot trobar més informació a [2], es basa en la interposició dinàmica de codi en la llibreria de Pthreads. D'aquesta manera és capaç d'obtenir informació sobre l'estat dels *threads* que es troben en el sistema, però no podia capturar dinàmicament dades relacionades amb l'estat intern de la JVM.

Aquesta versió de JIS va permetre començar a estudiar el comportament de la JVM en entorns paral·lels. Tot i la mancança d'informació que presenta pel que fa a dades de la JVM, aquestes van poder ser deduïdes estàticament (mirant-ne el codi) i així es va poder començar a analitzar el rendiment d'algunes aplicacions Java conegudes (principalment *benchmarks* numèrics).

Primera Versió de JIS basada en JVMPI

En un anterior projecte de final de carrera, realitzat per l'alumne Javier Bartolomé Rodríguez, es va realitzar el desenvolupament d'una versió de JIS basada en el JVMPI i que es trobava disponible per a les plataformes Linux (Intel), AIX (IBM) i SGI.

Aquesta versió presentava com a avantatge respecte a la versió nativa de SGI que representava molta informació relacionada amb l'estat de la Màquina Virtual de Java, ja que aquesta era provista pel JVMPI. La versió nativa (no basada en JVMPI) per a SGI no era capaç de mostrar aquestes dades.

Com a important inconvenient, aquesta versió de JIS presentava carències importants pel que fa a la informació que mostrava sobre l'estat dels *threads* a nivell de sistema, ja que el JVMPI no permet distingir quan un *thread* es troba corrent sobre un processador i quan es troba a punt per a córrer però el gestor de recursos del sistema no ha decidit assignar-li un processador.

El més important d'aquesta versió de JIS és que va introduir la recerca del grup BJS en els conceptes bàsics del JVMPI i va fixar les bases de coneixement utilitzades en aquest projecte per al desenvolupament de la nova versió de JIS.

Altres eines de *profiling*

En el mercat existeixen una certa quantitat d'eines pensades per al profiling d'aplicacions Java, algunes basades en el JVMPI i d'altres que utilitzen mètodes propis per a l'extracció de dades del sistema. La principal diferència entre la majoria d'aquestes i la filosofia que es troba darrere JIS és que no estan pensades per a treure el màxim rendiment de la plataforma sobre la que les aplicacions instrumentades s'executen.

En general, les aplicacions de profiling per a Java que existeixen estan pensades per a estudiar el comportament semàntic que aquestes tenen. És a dir, intenten fer un estudi del comportament de l'aplicació en temps d'execució que permeti als desenvolupadors detectar potencials problemes de programació. Així, habitualment ofereixen informació sobre en quines parts del codi es produeixen colls d'ampolla (quin bucle, quina rutina...).

JIS orienta la seva instrumentació a veure possibles patrons de degradació de rendiment en codis Java que s'executen en entorns, principalment, paral·lels. Queda clar que el codi del desenvolupador ha de ser eficient per a aconseguir que l'aplicació ho sigui. Però hi ha molts factors de rendiment que no són detectables a partir del codi font de l'aplicació, i se n'ha de buscar la raó en el funcionament de la màquina i del sistema operatiu.

En el report [1], es pot trobar una àmplia comparativa d'eines de *profiling* per a Java. Cadascuna d'elles és analitzada breument, i les diferències i similituds amb JIS (a part dels seus punts forts i punts febles) són comentades.

2.1.2 Requeriments

En aquest punt es presenten els principals requeriments que s'imposen a JIS en el seu desenvolupament. Aquests, es divideixen en tres grups, segons el punt del sistema que afecten.

La instrumentació d'aplicacions escrites en Java és un subconjunt de la instrumentació general de sistemes, amb la particularitat que l'aplicació estudiada funciona com si fos un sistema en ella mateixa. És a dir, la màquina virtual de Java intenta crear una abstracció per a les aplicacions que hi corren al damunt que les allunyi de les particularitats que presenta cadascun dels sistemes en què es puguin executar.

Es tracta, doncs, d'instrumentar tant els events que es donen al sistema on s'executa la màquina virtual de Java com els events que es donen dins de la màquina virtual. D'altra manera, difícilment serà possible comprendre el comportament d'una aplicació Java, ja que aquestes es troben condicionats per l'estat del sistema natiu com de l'entorn virtual proporcionat per la JVM.

Les aplicacions Java a estudiar, per tant, es poden considerar com a aplicacions que corren en tres nivells diferents: sistema natiu, sistema virtual i usuari. D'aquesta manera, una crida típica a sistema (com per exemple una escriptura a disc), realitzada des de codi Java, es descomposa en les tres següents etapes, tal com es mostra a la figura 2.1:

1. El codi d'usuari que realitza la crida,
2. El codi dins de la JVM que invoca les crides natives,
3. El codi dins del sistema natiu que realitza les operacions sobre el hardware

Aquesta arquitectura cal tenir-la en compte a l'hora de pretendre instrumentar el comportament de les aplicacions ja que, si es vol oferir una informació fidel al que ha succeït realment durant el temps d'execució, les interaccions d'aquestes tres capes són fonamentals.

D'eines d'instrumentació d'aplicacions Java n'hi ha força al mercat. En l'estudi fet a [1], són diverses les eines que s'estudien (a part de JIS) i totes elles presenten un element en comú que les fa poc útils de cara a l'estudi en profunditat d'aplicacions: obvien el nivell de sistema natiu. Les eines disponibles fins ara al mercat, se centren principalment en la vessant purament semàntica del comportament de les aplicacions: quines coses fan i com les fan, no a quin "ritme" es fan. Basant-se en un exemple clarificador, permeten saber que una iteració s'ha repetit n vegades, que cada iteració ha trigat x segons en executar-se, que el global de la iteració ha trigat $n{\cdot}x$ segons en completar-se i

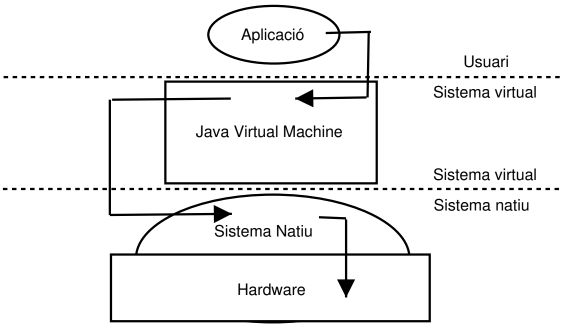

Figura 2.1: Estructura en tres nivells de les aplicacions Java

quines crides s'han fet dins de cada iteració. El que no permeten saber és què ha estat passant en el sistema durant aquests $n{\cdot}x$ segons. Podria ser que l'aplicació s'hagués passat la major part del temps bloquejada[1] esperant una lectura de disc o podria haver estat usant el 100% de la CPU que tenia disponible per a la realització de càlculs intensius.

Aquesta capacitat de distingir estats a nivell de sistema natiu es torna absolutament necessària des del moment que es pretén estudiar una aplicació a baix nivell. Aquesta situació es presenta moltes vegades quan de les aplicacions se n'estudia el rendiment ofert, i encara més quan les execucions es produeixen en màquines paral·leles d'altes prestacions de les quals es pretén explotar al màxim les capacitats.

Aquest últim entorn és el que té com a objectiu l'eina JIS-JVMPI desenvolupada: entorns paral·lels d'altes prestacions. Els punts bàsics a considerar en el desenvolupament de JIS, es presenten a continuació.

Nivell JVM

És necessari poder aconseguir informació de l'estat de l'aplicació a nivell intern de la JVM. La JVM, com a màquina virtual que és, té el seu quadre d'estats per a les aplicacions que hi corren al damunt. Per tal d'entendre aplicacions Java, cal estudiar com la JVM les està tractant internament i en quin estat es troba la JVM.

Nivell de sistema operatiu

La JVM dins del sistema és un procés més. Per tant, per a entendre com s'està comportant i per què ho està fent d'una determinada manera, és requisit indispensable conèixer com la màquina virtual s'està comportant enfront del sistema operatiu. Això permet tenir una visió molt més "crua" del comportament de l'aplicació Java. Cal mostrar-ne l'estat a nivell sistema, i fer després els esforços necessaris per a poder posar en relació aquesta informació amb l'obtinguda a nivell JVM.

---

[1] En aquest text es farà distinció, principalment, entre tres estats en els que es pot trobar un thread d'aplicació actiu al llarg de la seva vida:

- Running: El thread es troba corrent sobre una CPU
- Ready o idle: El thread es troba a punt per córrer, però el sistema no li ha assignat CPU
- Bloquejat: El thread es troba a l'espera d'algun event de sistema (entrada/sortida o adquisició d'algun *lock*) i no es troba en disposició de poder córrer

Generació de traces

El CEPBA disposa de l'eina Paraver per a la visualització i anàlisi (qualitatiu i quantitatiu) de tot tipus de dades generades a partir d'eines d'instrumentació. Com a eina d'ús genèric, dins del seu àmbit, té clarament especificat un format de dades propi.

JIS sempre ha estat orientat a la generació de traces visualitzables amb Paraver. Per aquesta raó és requisit indispensable de l'eina que el seu resultat últim sigui precisament això, una traça Paraver.

Instrumentació d'usuari des de codi Java

En moltes situacions, un desenvolupador d'aplicacions Java pot considerar necessari indicar el pas per un punt del codi determinat dins d'una traça Paraver generada per JIS. Aquest tipus de situacions, comporta un requeriment més per a JIS: ha de ser capaç d'oferir una interfície per a la inserció d'events, originats en el codi Java, dins les traces Paraver generades. Els events introduïts han de mantenir una consistència total amb la resta de la traça tant pel que fa a dades com pel que fa temps.

3 Desenvolupament tècnic

3.1 Desenvolupament de JIS-JVMPI per Linux IA32

3.1.1 Objectius pel que fa a la informació a instrumentar

Un cop presentades les necessitats de la creació d'una eina com JIS-JVMPI en el punt 2.1, és moment de veure amb més detall quin tipus d'informacions es volen obtenir de l'execució d'aplicacions java instrumentades.

Aquests objectius venen marcats per dos factors: les necessitats d'informació trobades pel grup de recerca en java (Barcelona Java Suite [7]) del CEPBA[6] durant les seves tasques de comprensió d'aplicacions java paral·leles en entorns multiprocessador, i les facilitats/limitacions presentades per l'eina de visualització de traces del CEPBA: Paraver[8].

Els objectius marcats pel que fa a informació són, doncs:

- Estat de cada *thread* Java[1]: *Run*, *Ready* o *Blocked*
- Cicle de vida de cada *thread* (inici i destrucció)
- CPU sobre la que cada *thread* corre en cada moment
- Noms[2] assignats per la JVM als *threads* instrumentats
- Inici i final de les operacions d'entrada/sortida
- Mida dels buffers tractats en les operacions d'E/S
- Canals sobre els que s'efectuen les operacions d'E/S
- Distinció entre operacions sobre canals d'E/S ordinaris i operacions sobre *sockets*

---

[1] La majoria de sistemes actuals donen suport per a tenir dos tipus de *threads* corrent en una aplicació: els *threads* de sistema i els *threads* d'usuari. Els primers són aquells que es corresponen a un *thread* de *kernel*. Els segons són una abstracció dels primers i permeten tenir *n threads* mapejats sobre un sol *thread* de *kernel*. Els primers permeten una millor gestió i rendiment, i els segons permeten funcionar millor les aplicacions en entorns que no suportin directament *threads* de sistema o en aquells en què les aplicacions utilitzin el suport *multithread* no per a millorar-ne el rendiment si no per una qüestió d'estructuració.

La màquina virtual de Java, en crear nous *threads* de java, pot funcionar segons els dos models anteriors. Permet crear *thread* natius (1 *thread* de java es correspon a un *thread* de *kernel*) o els anomenats *green threads* (*n threads* java es mapegen sobre un *thread* de sistema).

En entorns on el rendiment sigui un factor determinant, l'ús de *threads* natius és tècnicament obligatori. Per aquesta raó, les eines descrites en aquesta documentació (JIS-JVMPI en les seves diferents versions) només donen suport per a execucions que utilitzin *threads* natius.

Com a exemple de la situació actual en el món de les màquines virtuals de java pel que fa a la creació de *threads* java, la JVM de IBM (líder indiscutible de rendiment en les plataformes en què es troba disponible) ja només dona suport per a *threads* natius.

[2] La màquina virtual de Java anomena els *threads* internament segons uns noms definits, per l'usuari en els threads creats per aquest i prefixats en els *threads* que formen part de l'estructura interna de la màquina virtual. Aquests noms, però, no són visibles directament pel sistema. Es tracta, doncs, d'una mapeig que realitza la màquina virtual entre els identificadors de *thread* de sistema (normalment un nombre natural) i els noms que l'usuari coneix i li resulten familiar.

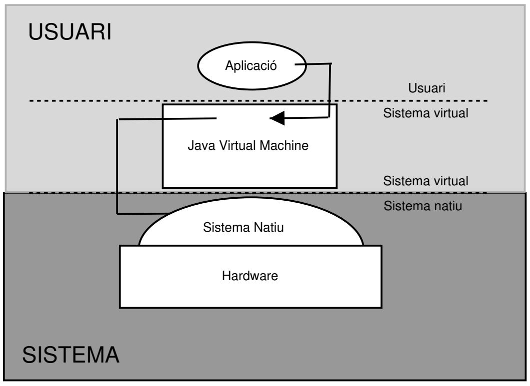

Figura 3.1: Arquitectura d'una aplicació java segons l'entorn d'execució

- Generació d'events d'usuari mitjançant una interfície comuna
- Informació sobre events interns de la JVM

3.1.2 Arquitectura de dos nivells

Tal com s'ha presentat en el punt 2.1, es pretén crear una eina que permeti estudiar les aplicacions Java des d'un punt de vista dividit en tres nivells: usuari, sistema virtual i sistema natiu. Per tal de donar resposta a aquests requeriments, cal analitzar què és necessari per a capturar informació d'aquests tres nivells.

El primer fet que salta a la vista en veure l'arquitectura en tres nivells presentada a la figura 2.1 és que des del punt de vista d'entorn d'execució són dos, i no tres, els nivells en què es desenvolupa una aplicació Java: usuari i sistema, tal com es mostra a la figura 3.1.

Aquests dos nivells són els únics que coneix el sistema natiu en què s'executa la màquina virtual de Java. El fet de voler extreure informació de l'espai tant de sistema com d'usuari, obliga a fraccionar l'arquitectura de JIS[3] en dos parts: una que s'executi amb privilegis de sistema i una que corri vinculada a l'aplicació a instrumentar.

Les dos parts que formen l'arquitectura de JIS són:

1. **Llibreria vinculada a l'aplicació instrumentada**: Es tracta de tenir un objecte de codi carregat dins del procés de la màquina virtual de java en el moment de l'execució de l'aplicació instrumentada per tal de capturar els events que s'hi provoquin i poder generar la informació necessària per a poder-ne generar una traça[4] visualitzable amb l'eina Paraver. Aquesta llibreria també serà utilitzada per a la generació d'events de JIS des del codi Java de l'aplicació, en cas que es consideri necessari.
2. **Mòdul de kernel i un dispositiu asociat**: Es tracta de tenir inserit un mòdul en el kernel, addicionalment a haver realitzat mínimes modificacions en el kernel de Linux que es presentaran més endavant, per tal de poder extreure informació de l'espai de sistema impossible d'aconseguir pels mitjans existents per defecte en el nucli de Linux sense realitzar-hi modificacions. El dispositiu és usat per a realitzar operacions de control sobre el mòdul i per a poder fer-ne un mapeig de la memòria en espai d'usuari. Aquests aspectes s'aniran describint seguidament en aquest apartat.

---

[3] En aquest punt JIS serà emprat com a abreujament de JIS-JVMPI per a Linux IA32

[4] El resultat d'un procés d'instrumentació, en l'entorn d'aquest projecte, serà sempre una traça en format natiu de Paraver. Per tant, sempre que s'esmenti el terme *traça*, serà referit a una traça Paraver generada per un procés d'instrumentació determinat.

Cal remarcar aquí, que **cadascuna de les dues parts que composen JIS disposa del seu propi buffer d'events independent**, un de nivell usuari dins de la llibreria i un de sistema en el mòdul de kernel. **Això és independent del mapeig que es fa del buffer de sistema en espai d'usuari** per raons que es detallaran en el punt 3.1.4.

Donat que la informació s'extreu des de dos orígens diferents, en algun moment s'haurà de dur a terme un procés de fusió de les dades capturades des d'ambdós punts. Això, a part de qüestions de rendiment, obliga a convertir la instrumentació de JIS en un procés de dos passades: primerament es genera la informació en suports separats segons l'origen de les dades i posteriorment (un cop acabada l'execució de l'aplicació) se'n fa una fusió i ordenació en una traça conjunta i definitiva.

3.1.3 Origen de les dades recollides

En el punt 3.1.1s'han presentat els requeriments d'informació que s'imposen a JIS. en aquest punt es pretén presentar els orígens d'aquestes dades, és a dir, quina part de JIS serveix per a extreure cada informació sobre el sistema. Els quatre punts d'origen de dades i la seva funcionalitat es presenten a continuació:

1. ***Patch* de kernel:**
    a) Extracció de l'estat dels *threads*
2. **Mòdul de kernel:**
    a) Intercepció de crides al sistema
        i. Operacions d'entrada sortida
        ii. Operacions sobre *sockets*
3. **Intefície JVMPI:**
    a) Inici i final de la Màquina Virtual de Java
    b) Creació i destrucció de *threads* de Java
4. **Crides natives des de codi Java:**
    a) Interfície genèrica per a la generació d'events d'usuari

L'estructura general de JIS per a donar resposta a aquestes generals es presenta en la figura 3.2. En ella es pot veure com interactuaran les diferents parts de la solució implementada, i com es repartiran la feina de recol·lecció de la informació.

Amb aquestes dades, els events que poden ser emmagatzemats en una traça Paraver des de JIS són:

- CONTEXT SWITCH: Canvi de context d'un thread
- SOCKET ACCEPT: Event d'*accept* d'una connexió entrant per part d'un *socket*
- SOCKET READ: Lectura de dades en un *socket*
- SOCKET WRITE: Escriptura de dades en un *socket*
- SOCKET SHUTDOWN: Destrucció d'un *socket*

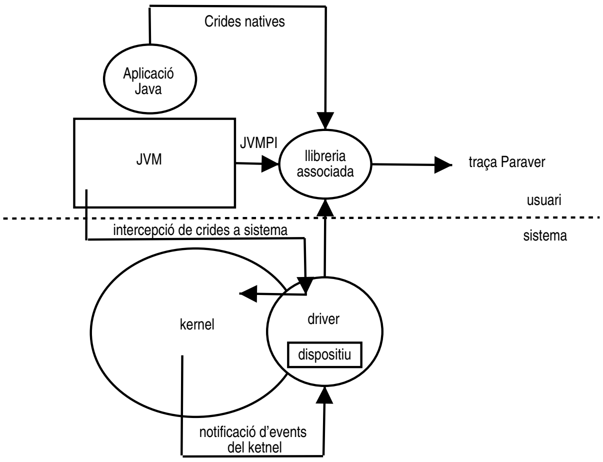

Figura 3.2: Esquema general de l'arquitectura de JIS

- SOCKET CREATE: Creació d'un *socket*
- IO READ: Operació de lectura sobre un canal
- IO WRITE: Operació d'escriptura sobre un canal
- IO CLOSE: Tancament d'un canal
- IO EVENT: Event d'entrada sortida[5]
- IO SIZE: Mida d'una operació d'entrada sortida[6]
- THREAD START: Inici d'un *thread* de Java
- THREAD END: Finalització d'un *thread* de Java

3.1.4 Accés a les dades de sistema

Linux presenta com a principal avantatge davant de molts altres sistemes operatius existents que el seu codi font és obert i molt estudiat, de manera que la opció de realitzar-hi certes modificacions de cara a adaptar-ne les seves característiques a les necessitats de cada projecte o usuari és una opció força viable, quan els requeriments ho fan aconsellable.

En el cas de JIS, la modificació del *kernel* de Linux ve donada per la necessitat de controlar en tot moment l'estat dels *threads* pertanyents a la màquina virtual de Java. El nucli de Linux ofereix certes utilitats (com ara el directori */proc*[7]). La limitació que presenten aquests mètodes pel que fa a l'extracció de l'estat dels *threads* del sistema és que considera equivalent els estats de *ready* i *run*, anomenant-los conjuntament *runnable*. Aquesta informació, sens dubte valuosa en moltes circumstàncies, no és ni de bon tros suficient per tal de caracteritzar el comportament d'una aplicació,

---

[5] Els events d'entrada/sortida es marquen amb un event d'aquest tipus amb valor 1 a l'inici i amb valor 0 en finalitzar. D'aquesta manera, usant les eines que Paraver proporciona, és relativament senzill delimitar la duració d'una operació d'aquest tipus.

[6] La mida del buffer associada a un event d'entrada/sortida s'inserta a la traça amb un event d'aquest tipus (definint la mida com a valor de l'event), que apareix en el mateix instant de temps que l'event d'entrada sortida amb valor 0 que indica el final de l'operació. Aquest mètode es deu a que Paraver només permet associar un valor a cada event.

[7] El directori *proc* de Linux, en cas de ser suportat per la versió de *kernel* utilitzada, consisteix en la creació d'un sistema de fitxers (*procinfo*) en què es pot trobar informació de moltes dades del sistema. Per exemple, es pot veure per cada procés del sistema en quin estat es troba.

i molt menys en un entorn multiprocessador, on el número de *threads* en estat de *run* (no *runnable*) en cada moment indica l'ús que es fa dels diferents processadors de la màquina en què s'instrumenta l'aplicació.

Paral·lelament a la modificació del *kernel* de Linux, per tal d'extreure altres informacions de l'espai de sistema necessàries per al procés d'instrumentació, durant l'execució de les aplicacions java s'intercepten determinades crides a sistema per tal d'instrumentar-les. Aquest procés es du a terme mitjançant la inserció d'un mòdul de *kernel* de Linux en el sistema i l'ús d'un dispositiu (amb el seu driver associat) per a accedir a les dades extretes.

Estructures de dades

La recollida de dades que es fa des del *kernel* necessita una zona de memòria per a dipositar-hi els events que es van detectant. En aquest punt es presenta l'estructura de dades encarregada de suport a aquest emmagatzemament de dades i l'ús que se'n fa.

L'estructura de dades per a guardar un event es pot veure en la figura 3.3. Es tracta de l'estructura *state_ record*. Els seus camps són:

- pid[8]: PID del thread associat a l'event capturat.
- time: *Timestamp* de l'event.
- cpu[9]: Processador associat a l'event.
- state: Estat en què es troba el thread (només vàlid per a events associats a un canvi d'estat d'un thread).
- socket_port: Canal associat a un event d'entrada/sortida.
- aux: Camp usat per a usos extres que es necessitin en determinats events.

Aquesta és doncs l'estructura utilitzada per a guardar informació d'un event individual. El conjunt de tots els events capturats i guardats es troba en l'estructura *threads_ data*, els camps de la qual són:

- buffer: El *buffer* d'events individuals emmagatzemats. Es tracta d'un buffer circular on es guarden els events generats en el *kernel* o arrel de la intercepció de crides a sistema per part del mòdul de *kernel*. La seva mida ve definida per la constant RECORDS_SIZE, que es troba definida en funció del número de pàgines de memòria que es volen consumir per l'estructura *threads_ buffer* i de la mida de la pàgina definida en el sistema.
- max: Límit superior del buffer circular.
- end: Posició del buffer en la qual es pot escriure el següent event. El buffer es troba ple quan es compleix que $end = max$
- num_cpus: Número de processadors disponibles en els sistema.

---

[8] PID (*Process Identifier*): Identificador d'un procés. En Linux, un thread es considera un procés lleuger de manera que se li assigna un identificador de procés. En d'altres sistemes, el procés té assignat un pid i dins del procés cada thread té un tid (*Thread Identifier*).

[9] No sempre és aplicable ja que hi ha events que no es troben associats a una CPU. Per exemple, quan l'event és un canvi d'estat del *thread* per indicar que aquest s'ha aturat, no hi ha CPU associada ja que un thread que no està corrent no es troba directament associat a cap processador.

- first_pgrp: Número de *pthreads group*[10] del procés que s'està instrumentant.
- free: Número de posicions lliures que es troben en el buffer.

Aquestes dues estructures de dades (i la gestió que porten associades) permeten mantenir emmagatzemats de manera estructurada els events que es capturen en el nucli de Linux de manera ràpida i eficient ja que es troba en memòria.

El *buffer* principal (el camp *buffer* de l'estructura *threads_ data*) es reserva en memòria quan es carrega el *driver* associat al dispositiu d'instrumentació, tal com es pot veure en l'algorisme 1. La crida *vmalloc* serveix per a reservar una zona contigua de memòria en l'espai d'adreces virtuals. Retorna una adreça virtual[11] de kernel, és a dir, les pàgines físiques reservades no tenen per què ser consecutives (a diferència del que fa *kmalloc*, que retorna adreces lògiques de kernel), però de cara a les adreces virtuals que genera el *kernel*, sí que ho són.

La memòria reservada es retorna al sistema quan el *driver* del dispositiu d'instrumentació es descarrega de memòria.

Algorisme 1 Reserva d'espai per al buffer de sistema

```
if(((void *)threads = vmalloc(PAGE_SIZE*NUM_PAGES)) == NULL)
{
   printk("<0>Error obtenint memoria\n");
   return -1;
}
```

Driver i dispositiu d'instrumentació

Com s'ha comentat anteriorment, l'arquitectura de la solució proposada per a capturar events de sistema es fonamenta en dues parts: unes modificacions en el kernel i la creació d'un dispositiu d'instrumentació amb el seu driver corresponent. En aquest punt es detalla el funcionament del dispositiu i del *driver* associat.

La raó principal que porta a la creació d'un dispositiu d'instrumentació és que en algun moment previ a la generació de la traça definitiva és necessari fusionar les dades extretes del nivell de sistema (i guardades en el buffer de sistema, dins del mòdul de kernel) amb aquelles recopilades des de l'espai d'usuari (i guardades en un buffer en l'espai d'usuari, a la llibreria compartida de JIS) . Per tal de dur a terme aquesta acció, és necessari poder accedir a les dades de nivell sistema des de l'espai d'usuari. Això és així perquè l'altra possibilitat (accedir a les dades d'usuari des de l'espai de sistema) no és viable ja que en temps d'execució de sistema les accions han de ser tan curtes com siguin possibles i el fet de fusionar i ordenar events per al final bolcar-los a disc és massa costosa.

La solució del dispositiu presenta dos punts positius importants:

---

[10] Pthreads (de *Posix Threads*) és una especificació estàndard de la interfície de la llibreria de *threads* d'usuari del sistema. Cada sistema pot tenir diverses llibreries de threads disponibles i totes aquelles que segueixin en la seva interfície de crides l'especificació *Posix* es consideren llibreries de *pthreads*. La majoria de sistemes (UNIX i no UNIX) tenen alguna implementació de *pthreads*.

El *pthreads group* serveix per a associar un conjunt de pthreads que mantenen alguna relació de creació. En el cas que ens ocupa, els *threads* associats dins del grup que es guarda en la variable first_pgrp són aquells que pertanyen a la màquina virtual de Java, i per tant són aquells que cal considerar a l'hora de capturar-ne events.

[11] El *kernel* pot treballar amb adreces lògiques i virtuals. Les virtuals són aquelles que es gestionen de la manera habitual (a través de la *Memory Management Unit* o MMU), tal com es fa amb les adreces dels processos d'usuari. Les lògiques són adreces que abracen gairebé tota la memòria física i que només es diferencien de les adreces físiques en què se'ls aplica un desplaçament constant per tal que les adreces referenciades internament no entrin en conflicte amb les adreces virtuals gestionades per la MMU.

```
#define NUM_PAGES 1024
#define RECORD_SIZE sizeof(state_record)
#define RECORDS_SIZE (((PAGE_SIZE * NUM_PAGES) - (5*sizeof(long))) /
                      RECORD_SIZE)

typedef struct state_record {
  unsigned int pid;
  unsigned long long time;
  unsigned char cpu;
  char state;
  unsigned long socket_port;
  unsigned long aux;
} state_record;

struct threads_data {
  unsigned int max;
  unsigned int end;
  unsigned int num_cpus;
  unsigned int first_pgrp;
  unsigned int free;
  state_record buffer[RECORDS_SIZE];
};
```

Figura 3.3: Estructures de dades per a la recollida d'events en espai de sistema

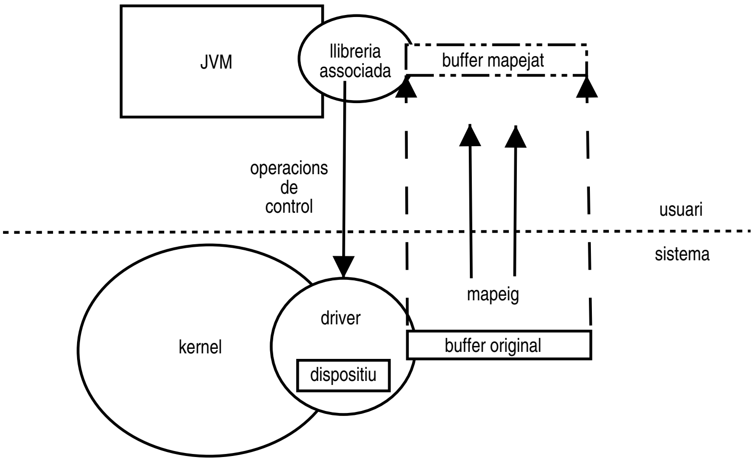

Figura 3.4: Esquema de la solució proposada pel nivell sistema

- Permet accedir al *buffer* de sistema (amb tots els seus events associats) simplement mapejant-lo en una zona de memòria de l'espai d'usuari. Un cop fet això, el codi d'usuari no nota cap diferència entre la memòria reservada directament i la mapejada des de l'espai de sistema
- Permet realitzar operacions senzilles sobre el dispositiu per tal d'exercir un cert nivell de control sobre la instrumentació de sistema que s'està realitzant. Per exemple, permet indicar quan començar i aturar la recollida d'events. Aquestes operacions han estat implementades a través de la operació ***ioctl***, que permet passar alguns paràmetres al dispositiu, per tal de provocar algunes operacions sobre el *driver* que el gestiona. El codi d'aquestes operacions es pot veure en l'algorisme 2.

Algorisme 2 Rutina ioctl del dispositiu

```
int m_ioctl (struct inode *s_inode,
             struct file *s_file,
             unsigned int cmd,
             unsigned long arg) {
  switch (cmd) {
    case STOP_TRACING:
      clear_jis_buffer();
      break;
    case END_DUMPING:
      dumping = 0;
      break;
    default:
      break;
  }
  return 0;
}
```

En la figura 3.4 es pot veure l'esquema general de la solució proposada.

Tot dispositiu creat en el sistema requereix un *driver* associat que n'implementi les operacions i que n'exerceixi el control necessari per tal de fer-lo funcionar correctament. En Linux, els drivers es poden implementar a través de dues formes diferents:

```
$ mknod /dev/jis c 245 0
```

Figura 3.5: Comanda per a la creació del dispositiu d'instrumentació

```
$ insmod driver_jis.o
```

Figura 3.6: Comanda per a la inserció del mòdul de *kernel* de JIS en el sistema

- Introduint el codi font del driver en el codi font del nucli i compilant de nou aquest
- Creant un *Linux Kernel Module*[12] (o LKM)

En el cas d'aquest projecte, la solució adoptada ha estat la creació d'un mòdul de *kernel* ja que així el codi font del kernel resulta més compacte.

El dispositiu creat s'anomena *jis,* i en el cas de Linux es crea segons la comanda que es mostra en la figura 3.5. En ella es pot veure com es crea un dispositiu de nom *jis* en el directori */dev*[13]. Aquest es crea de mode caràcter[14] amb la opció "c", amb *major*[15] 245 i *minor* 0.

Un cop es troba creat el dispositiu, és el moment de registrar el driver que el gestionarà dins del sistema. Donat que el driver es troba dins d'un mòdul de *kernel* creat, el primer pas a fer és inserir el mòdul en el sistema. Això es fa amb la comanda mostrada a la figura 3.6. La comanda ha de ser executada com a administrador del sistema (*root*) i rep com a paràmetre el nom del fitxer objecte que implementa el mòdul. En cas de voler eliminar el mòdul del sistema es pot executar la comanda mostrada en la figura 3.7. Finalment, per a veure que el mòdul es troba correctament inserit en el sistema, és necessari executar la comanda que es mostra, juntament amb la seva sortida, en la figura 3.8.

Un cop el mòdul s'insereix en el sistema, es crida la funció *init_module* d'aquest, on s'han de realitzar les accions pertinents per a registrar el *driver* que el mòdul implementa per tal que se'l notifiqui de les operacions a realitzar sobre el dispositiu creat. Això es fa amb el codi que es presenta en l'algorisme 3. En ell es poden veure:

- Estructura de dades *file_operations* on es guarden els punters a les funcions que implementen les diferents operacions suportades pel dispositiu
- Operació de registre del dispositiu, fixant el *major* al que respon, passant-li l'estructura d'operacions definida i definint un nom intern per al dispositiu.

També dins del codi d'inserció del mòdul de *kernel* en el sistema es realitza la reserva d'espai per al buffer de sistema, tal com es pot veure en l'algorisme 1.

Amb aquest pas, el *driver* es troba en disposició de rebre les diferents peticions del sistema. Per tal de poder donar resposta a aquestes accions, cal implementar les operacions que es pretén que

---

[12] Un mòdul de *kernel* de linux és un fitxer objecte que s'enllaça amb el codi del nucli en temps d'execució i, en general, sota demanda. És a dir, es pot disposar d'un *kernel* relativament petit però amb suport per a moltíssims dispositius a través de mòduls de *kernel*.

[13] En Linux, els dispositius del sistema s'acostumen a ubicar en el directori */dev*

[14] Els dispositius de linux poden ser de tipus bloc o de tipus caràcter. En general, els de tipus bloc s'utilitzen per a dispositius d'emmagatzemament massiu de dades i els de caràcters per a la resta. La principal diferència entre ells radica en la unitat mínima de treball que presenten: uns treballen sempre sobre blocs de dades (agregats de la unitat de dades) i els altres, els de caràcters, treballen directament sobre les unitats de dades.

[15] Tot dispositiu porta associat un nombre anomenat *major* i un altre anomenat *minor*. El *minor* és feble respecte al *major*. El *major* identifica una família de dispositius (per exemple, discos SCSI) i el *minor* n'identifica el model dins de la família. En el cas de jis, s'utilitza el 245 perquè és un nombre no utilitzat per defecte i conegut.

```
$ rmmod driver_jis
```

Figura 3.7: Comanda per a la eliminació del mòdul de *kernel* de JIS del sistema

```
$ lsmod
```

Module	Size	Used by
vmnet	18304	6
vmmon	18816	0
driver_jis	4880	0

Figura 3.8: Comanda per al llistat dels mòduls que es troben inserits en el sistema

Algorisme 3 Registre del driver de jis i estructura d'operacions implementades

```
const struct file_operations fops = {
      NULL, /* owner */
      NULL, /* lseek */
      NULL, /* read */
      NULL, /* write */
      NULL, /* readdir */
      NULL, /* poll */
      m_ioctl, /* ioctl */
      m_mmap, /* mmap */
      m_open, /* open */
      NULL, /* flush */
      m_close, /* release */
      NULL, /* sync */
      NULL, /* async */
      NULL, /* check media change */
      NULL, /* revalidate */
      NULL, /* lock */
};
if((driver_major = register_chrdev(JIS_MAJOR, "jis_driver", &fops)) < 0)
{
    printk("<0>Impossible to register JIS driver with major JIS_MAJOR.
            Please, consult the BJS team.\n");
    return -1;
}
```

suporti. Les dues principals operacions implementades són la d'obertura i tancament del dispositiu. L'algorisme 4 mostra el codi de la rutina d'obertura del dispositiu. L'algorisme 5 mostra la rutina de tancament del dispositiu.

Algorisme 4 Rutina d'obertura del dispositiu de traceig

```
int m_open (struct inode *s_inode,
            struct file *s_file)
{
  if(obert) {
    printk("<0> JIS driver already opened\n");
    m_close(NULL, NULL);
    return -1;
  }
  obert = 1;
  dumping = 0;
  MOD_INC_USE_COUNT;
  threads->first_pgrp = current->pgrp;
  threads->buffer[0].pid = current->pid;
  threads->buffer[0].time = get_cycles();
  threads->free = RECORDS_SIZE;
  threads->end = 1;
  threads->max = RECORDS_SIZE - 1;
  set_jis_buffer((struct threads_data *)threads, current);
  trace_stopped = 0;
  orig_sys_socketcall = sys_call_table[__NR_socketcall];
  sys_call_table[__NR_socketcall] = jis_sys_socketcall;
  orig_sys_read = sys_call_table[__NR_read];
  sys_call_table[__NR_read] = jis_sys_read;
  orig_sys_write = sys_call_table[__NR_write];
  sys_call_table[__NR_write] = jis_sys_write;
  orig_sys_close = sys_call_table[__NR_close];
  sys_call_table[__NR_close] = jis_sys_close;
  return 0;
}
```

En la rutina d'obertura del dispositiu es poden observar diverses seccions de codi:

- Inicialització de les variables de control de l'estructura de dades encarregada d'allotjar el buffer d'events de sistema. En aquest punt, és interessant observar la inicialització del *pthread_ group* i del temps d'inici del procés d'instrumentació.
- Assignació del buffer de sistema a les rutines d'extracció d'events de sistema incorporades directament en el nucli de Linux (aquest punt s'explicarà més endavant, en el punt 3.1.4).
- Inserció dels enllaços per a la intercepció de determinades crides a sistema (explicat més endavant, en el punt 3.1.4).

Pel que fa a la rutina de tancament del dispositiu, es pot observar:

- Es desenllaça el buffer d'events capturats de sistema de les rutines inserides directament dins del *kernel*.

Algorisme 5 Rutina de tancament del dispositiu de traceig

```
int m_close (struct inode *s_inode,
             struct file *s_file)
{
  if(obert) {
    clear_jis_buffer();
    trace_stopped = 1;
    obert = 0;
    sys_call_table[__NR_socketcall] = orig_sys_socketcall;
    sys_call_table[__NR_read] = orig_sys_read;
    sys_call_table[__NR_write] = orig_sys_write;
    sys_call_table[__NR_close] = orig_sys_close;
    threads->end = 0;
    threads->first_pgrp = -1;
    MOD_DEC_USE_COUNT;
    return 0;
  }
  else return -1;
}
```

- Es desfan les intercepcions de les crides a sistema escollides.

Mapeig del buffer

Com s'ha comentat en el punt anterior, el buffer de sistema és utilitzat des de l'espai d'usuari per tal de fer la costosa operació de fusió d'events capturats a nivell usuari (guardats al buffer de la llibreria compartida de JIS) i a nivell de sistema (guardats en el buffer del mòdul de kernel) en la traça final. També s'ha comentat que aquest accés es fa a través d'un mapeig de memòria entre els dos espais.

El mapeig és possible a través de la implementació de l'operació *mmap* en el *driver* del dispositiu d'instrumentació. El codi d'aquesta rutina es pot veure a l'algorisme 6. Primerament s'observa que el que fa la crida *mmap* en la seva implementació és senzillament marcar la regió de memòria d'espai d'usuari en la que es vol fer el mapeig com a *RESERVED* i es defineixen les operacions que l'han de gestionar. Això és així perquè es tracta d'un sistema de paginació sota demanda en el qual cada pàgina només es porta a memòria física quan es produeix una fallada de pàgina. Per tant, en fer el mapeig no és necessari fer cap mapeig real de pàgines. Només quan es produeixin fallades en l'espai d'usuari es realitzaran les modificacions necessàries per a fer que les pàgines lògiques sol·licitades d'espai d'usuari apuntin a les mateixes pàgines físiques que les pàgines lògiques d'espai de sistema que conformen el buffer. La implementació de les operacions necessàries per a la tasca pretesa passa només per escriure la funció *nopage* per a la regió de memòria. Aquesta rutina es crida automàticament quan es produeix una fallada de pàgina, moment en el qual és necessari realitzar el mapeig de memòria corresponent. El procés exacte es pot veure a la figura 3.9. El flag *RESERVED* fa que la regió de memòria modificada no pugui patir un procés de *swap out* i per tant sempre s'accedeix a memòria física. El que fa l'algorisme *nopage* és bàsicament afegir una entrada a la taula de pàgines d'usuari que apunti a la mateixa regió de memòria física que ho fa la pàgina de sistema.

Captura d'events per intercepció de crides al sistema

Determinats events de sistema han de ser capturats a través de la intercepció de les crides a sistema que els produeixen. En Linux això s'aconsegueix introduint en el vector de crides a sistema punters

Algorisme 6 Rutines de mapeig de memòria del dispositiu

```
const struct vm_operations_struct jis_vm_ops =
{ nopage: jis_vma_nopage, };
int m_mmap (struct file *file,
            struct vm_area_struct *vma)
{
  vma->vm_ops = &jis_vm_ops;
  vma->vm_flags |= VM_RESERVED;
  return 0;
}
struct page *jis_vma_nopage(struct vm_area_struct *vma,                             unsign
                            int write)
{
  unsigned long offset;
  struct page *page = NOPAGE_SIGBUS;
  void *pageptr = NULL;
  pgd_t *pgd;
  pmd_t *pmd;
  pte_t *pte;
  unsigned long lpage;
  offset = address - vma->vm_start + (vma->vm_pgoff << PAGE_SHIFT);
  pageptr = ((char *)threads + offset);
  lpage = VMALLOC_VMADDR(pageptr);
  spin_lock(&init_mm.page_table_lock);
  pgd = pgd_offset(&init_mm, lpage);
  pmd = pmd_offset(pgd,lpage);
  pte = pte_offset(pmd,lpage);
  page = pte_page(*pte);
  spin_unlock(&init_mm.page_table_lock);
  return page;
}
```

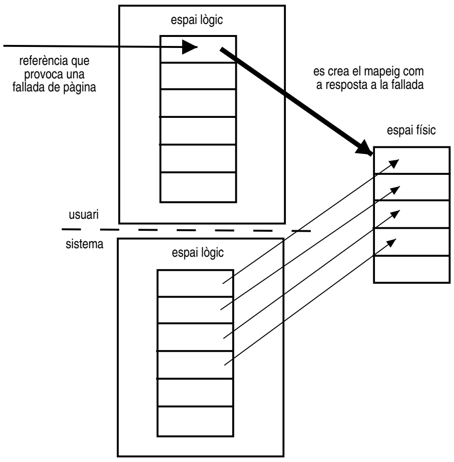

Figura 3.9: Procés de mapeig de memòria sota demanda

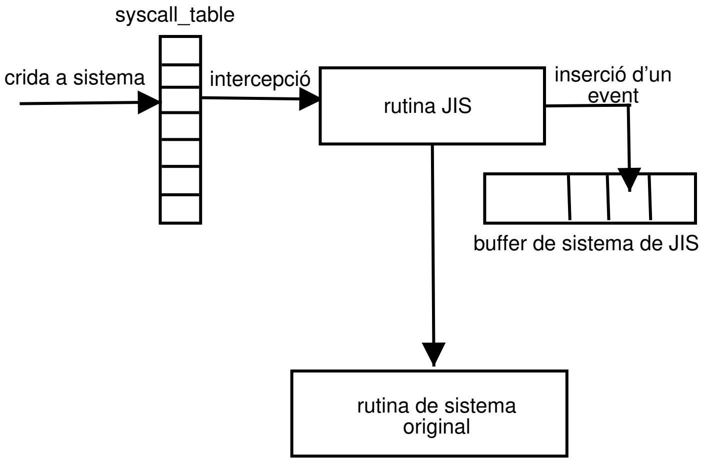

Figura 3.10: Esquema d'intercepció de crides a sistema

Nom	Funcionalitat
socketcall	rutines relacionades amb operacions sobre sockets
read	operacions d'entrada sortida: lectura
write	operacions d'entrada sortida: escriptura
close	tancament d'un canal

Taula 3.1: Crides a sistema interceptades

a funcions pròpies que finalment acabaran cridant a les rutines de sistema originals. Un esquema es mostra a la figura 3.10.

Les rutines interceptades fins el moment són les reflectides en la taula 3.1 . El procés d'intercepció d'aquestes crides es produeix en les operacions d'obertura i tancament del dispositiu, tal com es pot veure en els algorismes 4 i 5 respectivament. En ells es pot veure com es modifiquen determinades entrades de la taula global *sys_ call_ table* per a fer que en produir-se les corresponents crides a sistema sigui el codi propi, i no l'original de sistema, el primer a ser executat. Posteriorment en les crides es realitzarà la crida a les rutines originals per tal de no perdre funcionalitat en el sistema.

Un cop interceptada una crida a sistema (se'n pot veure un exemple a l'algorisme 7), el procediment a seguir consisteix en inserir un event en el buffer de sistema (cal recordar que aquesta interposició de codi es produeix des del mòdul de kernel, és a dir, des de l'espai de sistema). Primer però, cal fer algunes comprovacions:

- Comprovació que el buffer(variable *threads*) de sistema es troba inicialitzat (quan el dispositiu no es troba obert, el driver segueix inserit en el sistema però el punter del buffer és nul, de manera que no es guarden nous events).
- Comprovació que no s'ha aturat el procés d'instrumentació (variable *trace_ stopped*) a través d'una operació *ioctl* (de la qual se'n parla en el punt 3.1.4) sobre el dispositiu.
- Comprovació que el *pthreads group* del thread que genera l'acció és el del procés que s'està instrumentant

Amb aquestes dades es decideix si l'event ha de ser registrat o no. D'aquesta manera s'eviten posteriors filtres de dades i es redueix al mínim imprescindible les dades registrades. En cas que alguna de les comprovacions falli, es retorna directament el resultat de realitzar la crida a sistema original, de manera que no se'n modifica el comportament. En cas que les comprovacions indiquin que s'ha de registrar l'event, es crida una operació de registre (pròpia de cada crida a sistema interceptada) que s'encarrega de registrar l'event.

La crida encarregada de registrar l'event també ha de realitzar certes verificacions. Primer de tot cal tornar a comprovar que no s'ha procedit a aturar el procés d'instrumentació de l'aplicació i que el buffer de sistema és correcte. Un cop superades aquestes dues verificacions és moment d'introduir l'event en el buffer. Per tal d'evitar bloquejos innecessaris en el codi del *kernel*, els accessos a les posicions del buffer es realitzen mitjançant operacions atòmiques. Principalment, per adquirir el valor del comptador que indica la primera operació lliure en el buffer. La implementació de l'operació atòmica *fetch_ and_ inc* utilitzada, es pot veure en l'algorisme 8. Aquesta operació s'encarrega de fer una operació atòmica per tal d'obtenir el valor d'una posició de memòria i incrementar-ne el valor posteriorment, sense possibilitat que un altre *thread* en el sistema pugui accedir-hi. Un cop obtinguda la posició del buffer on registrar el sistema, tot el treball d'inserir l'event ja no està sotmesos a riscos derivats de *race conditions*[16], ja que l'apuntador a la primera posició lliure del buffer ja ha quedat modificat.

Un cop obtinguda la posició del buffer en què s'ha de registrar l'event, només manca introduir les dades correctes en el buffer i retornar. La majoria de dades que s'introdueixen s'extreuen directament de paràmetres de la crida a sistema interceptada. Només un s'extreu en temps d'execució: el *time stamp* de l'event. Sobre aquest tema es parlarà en el punt 3.1.7.

Algorisme 7 Exemple d'instrumentació d'una crida a sistema (read)

```
asmlinkage ssize_t jis_sys_read(unsigned int fd,
                                char *buff,
                                size_t count)
{
  long res;
  if(threads == NULL || trace_stopped ||
     current->pgrp != threads->first_pgrp || threads->free <= 0)
    return orig_sys_read(fd, buff, count);
  register_io_read(current->pid, current->processor, fd);
  res = orig_sys_read(fd, buff, count);
  return res;
}
void register_io_read(unsigned int pid,
                      unsigned char cpu,
                      long fd)
{
  long index;
  if(threads == NULL || trace_stopped || threads->free <= 0)
    return;
  index = fetch_and_inc(&threads->end) % RECORDS_SIZE;
  threads->buffer[index].pid = pid;
  gettime(&threads->buffer[index].time);
  threads->buffer[index].state = JIS_IO_READ;
  threads->buffer[index].cpu = cpu;
  threads->buffer[index].socket_port = fd;
}
```

---

[16] Aquell tipus de situacions que poden provocar problemes de concurrència i que no es presenten sempre de la mateixa manera ja que es troben condicionades per moltes variables d'execució.

Algorisme 8 Operació atòmica *fetch_ and_ inc*

```
static __inline__ unsigned int fetch_and_inc(unsigned int *v)
{
  unsigned int a = 1;
  __asm__ __volatile__( "lock;
                       xaddl %1,%0" :
                             "=m" (*v), "=r" (a) :
                             "r" (a)
                      );
  return a;
}
```

Modificacions en el kernel

L'extracció de dades de sistema realitzada interceptant crides de sistema que s'ha presentat en l'anterior punt, per a ser útil de veritat, ha de ser complementada amb informació sobre l'estat en tot moment dels *threads* que conformen l'aplicació instrumentada.

Linux ofereix com a interfície d'extracció de dades de sistema un sistema de fitxers especial: el *proc filesystem*. Consisteix en un directori (anomenat /proc) on es poden trobar subdirectoris per a cada procés del sistema, dintre dels quals hi ha fitxers amb informació sobre el procés a què referencien. Aquests fitxers no existeixen com a tals, si no que realitzen consultes des de l'espai d'usuari en temps d'execució.

El sistema de fitxers proc, tot i la seva indiscutible utilitat, no resultava suficient per a les pretensions pel que fa a informació extreta que presentava JIS. La principal limitació venia donada per la diferenciació d'estats que fa sobre els threads. Per al *proc filesystem*, els estats bàsics d'un *thread* poden ser:

- Runnable: Preparat per córrer en una CPU o ja corrent en una CPU
- Blocked: Bloquejat per decisió de l'aplicació
- Stopped: Parat

El problema apareix en el primer punt. Un estat runnable és molt poc informatiu de cara a entendre el comportament d'una aplicació, ja que és un multiestat que no defineix exactament què està succeint en el sistema. Els requeriments imposats per una eina com JIS obligaven a determinar amb precisió si un *thread* realment disposava de CPU o no en cada moment.

Donat que la plataforma en la que es plantejava l'eina, és a dir Linux, és de caràcter obert, es va decidir que una manera fiable i relativament poc complicada d'obtenir la informació era introduir codi en punts concrets del nucli per tal de registrar els events que s'hi produeixen. També era necessària una política de coordinació amb el mòdul de *kernel* i amb la part de l'instrumentador que es troba en espai d'usuari per tal de fer que els tres funcionessin coordinadament.

L'objectiu del *patch* del nucli de Linux era poder obtenir una informació més detallada sobre l'estat dels *threads* del sistema en tot moment. La informació que es pretenia aconseguir és:

- Run: Corrent en una CPU
- Ready: Preparat per córrer però sense CPU disponible (és a dir, bloquejat per decisió del sistema, no de l'aplicació)
- Blocked: Bloquejat per decisió de l'aplicació

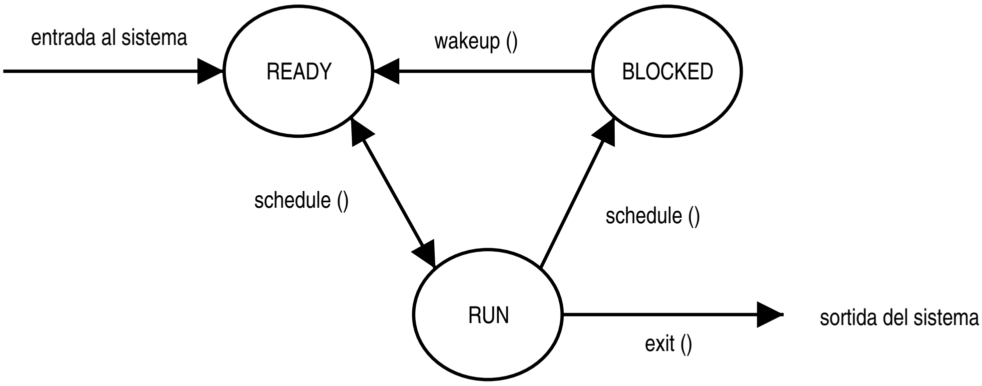

Figura 3.11: Diagrama d'estats dels processos a Linux i rutines que permeten detectar-ne les transicions

- Stopped: Parat

Per tal d'aconseguir aquestes dades, s'havien d'interceptar tots aquells punts en què es poden produir canvis d'estat en el processos[17] de Linux. Això, finalment passa per tres rutines concrets del codi del nucli:

- schedule() : És el cor de Linux ja que és on es realitza tota la planificació de processos. S'encarrega de decidir quins processos corren en cada moment i sobre quina CPU ho fan. Aquí també es pot detectar si el procés que deixa la CPU ho fa en estat de bloqueig o per decisió del sistema.
- exit () : Es crida quan un procés finalitza.
- wakeup () : Utilitzada per a despertar un procés que es trobava bloquejat esperant l'arribada d'un event concret.

En totes aquestes rutines es troben incorporades crides que, en cas de ser apropiat, es dediquen a registrar els events que s'hi produeixen. En la figura es pot veure l'esquema d'estats bàsic dels processos en Linux, i les rutines des d'on són detectades les transicions.

3.1.5 Ús del JVMPI

Totes les dades que s'obtenen directament de sistema són suficients per a poder estudiar amb molt de detall el comportament d'una aplicació. L'únic problema que es presenta és semàntic. El desenvolupador que estudia les seves aplicacions amb JIS necessita poder establir enllaços entre allò que ell coneix, l'aplicació a alt nivell, i com el sistema denomina internament els mateixos recursos. Aquesta

---

[17] En les actuals versions de Linux, el sistema utilitzat genèricament per implementar l'ús de *threads* no fa distinció entre aquests i els processos. Això es deu, almenys en el món de Java, a que la implementació de threads que s'està usant, els anomenats *Linuxthreads,* utilitzen la crida a sistema *clone* de Linux per a crear nous *threads.* Aquesta implementació, conforme a l'estàndard de *POSIX threads* (coneguts com a *pthreads*) que és l'utilitzat per la JVM, es basa en aquesta crida, la qual, clona un procés existent, però no en la seva totalitat ja que manté prou elements compartits (espai d'adreces i taula de canals, per exemple) per aconseguir que els dos "processos lleugers" es comportin realment com a *threads.* La qüestió està en que el sistema no assigna identificadors especials a tots els *threads* (clàssicament anomenats TID o *thread identifier*) si no que els assigna un PID (o *process identifier*).

La implementació de *pthreads* de Linux presenta una particularitat curiosa pel que fa a la sincronització entre *threads*: utilitza un *thread* extra, anomenat *ThreadsManager* que serveix per a sincronitzar i passar informació entre *threads* a través d'un sistema de *pipes.* Aquest *thread* és visualitzat en algunes de les traces que es poden observar en aquest document.

```
$ java -Xrunjis [nom_aplicació]
```

Figura 3.12: Invocació de la màquina virtual de Java per a usar JIS (considerant que JIS es troba en una llibreria compartida anomenada libjis.so i accessible per la JVM)

barrera es supera en JIS gràcies a la integració de les dades obtingudes sobre l'aplicació java a través del JVMPI en la informació produïda .

Per a funcionar, el JVMPI requereix l'ús d'una llibreria compartida en què s'introdueixin una sèrie de rutines predefinides. Aquesta llibreria serà passada com a paràmetre a la màquina virtual de Java en el moment de la seva invocació per tal que aquesta la carregui i la invoqui com li correspon.

En el moment de carregar la màquina virtual de java, amb la comanda mostrada en la figura 3.12 , la JVM crida a la rutina *JVM_ OnLoad* de la llibreria compartida, mostrada pel cas de JIS a l'algorisme 9 . En ella es pot veure la crida d'inicialització del mòdul de sistema, l'activació de la notificació de determinats events de la màquina virtual a través del JVMPI i, el més important de tot, s'indica a la JVM quina funció de la llibreria ha de cridar cada cop que es produeixi un dels events activats per a la notificació (en el cas de JIS, la rutina a invocar s'anomena notifyEvent).

Algorisme 9 Funció cridada per la JVM a la llibreria compartida de JIS en el moment de la càrrega

```
JNIEXPORT jint JNICALL JVM_OnLoad(JavaVM *jvm,
                                  char *options,
                                  void *reserved)
{
  int res;
  res=(*jvm)->GetEnv(jvm,(void**)&jvmpi_interface,JVMPI_VERSION_1);
  if(res<0)
    return JNI_ERR;
  jvmpi_interface->NotifyEvent=notifyEvent;
  inicializar_timer();
  if(initialize_module() < 0)
    return;
  CALL(EnableEvent)(JVMPI_EVENT_JVM_INIT_DONE,NULL);
  CALL(EnableEvent)(JVMPI_EVENT_JVM_SHUT_DOWN,NULL);
  CALL(EnableEvent)(JVMPI_EVENT_THREAD_START,NULL);
  CALL(EnableEvent)(JVMPI_EVENT_THREAD_END,NULL);
  jvmpi_interface->EnableGC();
  return JNI_OK;
}
```

El funcionament del JVMPI consisteix en què cada cop que es produeix un dels events activats dins de la JVM, realitza una crida a la funció de notificació d'events que ha estat seleccionada en temps d'inicialització, passant com a paràmetres determinades informacions sobre l'event produït.

En el cas de JIS, els events mínims a ser activats per tal que el sistema extregui la informació correctament són:

- JVM Init Done
- JVM Shut Down
- Thread Start

```
typedef struct {
  jint event_type; /* event_type */
  JNIEnv *env_id; /* env where this event occurred */
  union {
    struct {
      char *thread_name; /* name of thread */
      char *group_name; /* name of group */
      char *parent_name; /* name of parent */
      jobjectID thread_id; /* id of the thread object */
      JNIEnv *thread_env_id;
    } thread_start;
    ...
  } u;
} JVMPI_Event;
```

Figura 3.13: Definició abreujada del tipus JVMPI_Event

- Thread End

Aquests permeten, per una banda fer les operacions necessàries sobre la llibreria en iniciar i finalitzar l'execució de la màquina virtual, i per altra fer un seguiment del cicle de vida dels *threads* de Java en el sistema. Especialment important és l'event d'inici d'un *thread* ja que incorpora entre les dades notificades el nom intern del thread en la JVM. Això possibilita associar identificadors de *thread* de sistema (PID) amb els noms de *thread* que els desenvolupadors del codi Java assignen en temps de desenvolupament als threads que es creen. Aquesta aproximació entre les dades que es podrien anomenar "crues" de sistema i les dades especificades pels creadors de les aplicacions possibilita una major comprensió d'allò que es visualitza en estudiar una traça generada per JIS amb Paraver.

Quan un event és rebut a través del JVMPI, el primer que es fa és determinar de quin event es tracta. Això ho possibilita el paràmetre rebut per la funció *notifyEvent* associada a la notificació d'events pel JVMPI. El paràmetre és de tipus *JVMPI_Event*, la definició del qual es pot veure, de manera abreujada, a la figura 3.13 . En ella s'observa com, per l'event d'inici de *thread* es reben diferents informacions (entre elles el nom de l'esmentat *thread*). En l'algorisme 10 es pot veure l'estructura bàsica de la funció notifyEvent. El cos comprèn una distinció entre els diferents tipus d'events a tractar i una crida a la funció bàsica que n'implementa lacció associada.

En el cas de tractar-se d'un event d'inici de thread, el procediment seguit consisteix en:

1. Determinar el PID del *thread* creat (utilitzant la crida a sistema *getpid*)
2. Extreure el nom del *thread* i guardar-lo en un *array* de noms de *thread* per a poder extreure'l en el moment de generació de la traça final
3. Inserció d'un event de creació de thread en el buffer d'usuari[18]

Amb aquests tres procediments ja es té emmagatzemada prou informació sobre el *thread* per a poder generar la traça final de manera comprensible per als dissenyadors d'aplicacions Java i també suficient per a tornar a identificar el *thread* en propers events que generi.

En el cas de l'event de finalització de la màquina virtual, es procedeix a:

---

[18] El buffer d'usuari que es troba implementat presenta la mateixa estructura que el presentat en el punt .

Algorisme 10 Funció notifyEvent abreujada

```
void notifyEvent (JVMPI_Event *event) {
  ...
  switch(event->event_type) {
    case JVMPI_EVENT_JVM_SHUT_DOWN:
      get_time(&end_time);
      stop_tracing();
      parse_module_data(tiempo_fin);
      close_module();
      free_memory();
      CALL(ProfilerExit)((jint)-1);
      break;
    case JVMPI_EVENT_THREAD_START:
      Thread_start(event);
      break;
    case JVMPI_EVENT_THREAD_END:
      Thread_end(event);
      break;
    ...
 }
}
```

1. Obtenir el temps de final de traça
2. Aturar el procés d'instrumentació de sistema (el que es duu a terme en el mòdul de kernel i en el propi kernel) amb una crida *ioctl* embolcallada en la funció *stop_ tracing*
3. Inici del procés de fusió dels dos buffers existents, sistema i usuari en una traça final visualitzable amb Paraver
4. Tancament del dispositiu d'instrumentació i operacions paral·leles necessàries (bloqueig del buffer de sistema per tal d'evitar que es continuïn considerant possibles events)
5. Alliberament de memòria dels buffers d'usuari (els de sistema es mantenen reservats)
6. Sortida de l'instrumentador, permetent així al procés de la JVM a descarregar la llibreria compartida de memòria i poder descarregar-se ella mateixa.

Amb aquests passos el procés d'instrumentació de l'aplicació Java estudiada així com el procés de la pròpia JVM es poden donar per acabats. El resultat obtingut és una traça visualitzable amb Paraver en el directori d'execució.

L'event de final de *thread* només s'utilitza per a marcar un *thread* de Java com a finalitzat, de manera que properes aparicions del mateix *thread* de sistema amb nous events de Java es considerarien pertanyents a un altre *thread* de Java corrent sobre un mateix *thread* de sistema. Aquesta situació només es considerable en cas d'usar *green threads* com a model d'execució de la JVM enlloc de *native threads*.

L'event d'inici de la JVM es considera de cara a poder afegir-hi funcionalitats en el futur, però en la versió actual de JIS no es tracta de cap manera. Les inicialitzacions necessàries es realitzen en temps de càrrega de la JVM, de manera que un cop aquest procés ha finalitzat (moment en que es notifica l'event *JVM_ init_ done*) no és necessari preparar res per al funcionament de JIS.

```
typedef struct traza_t {
  tiempo_t tiempo;
  int tipo;
  int num_thread;
  int cpu;
  int pid;
  union {
    ...
    struct {
      char *nombre_thread;
      char *grupo_thread;
      char *thread_padre;
      long thread_id;
    } thread_start;
    struct {
      long thread_id;
    } thread_end;
    ...
  }u;
} traza_t;
```

Figura 3.14: Estructura abreujada d'un element del buffer d'usuari

Altres events de la JVM queden disponibles per a possibles implementacions de JIS però no són indispensables per al correcte funcionament de l'eina.

Buffer d'usuari

Respecte al buffer on s'emmagatzemen els events d'usuari, cal a dir que és diferent de l'utilitzat en l'espai de sistema. L'estructura bàsica que l'implementa és la que es presenta a la figura 3.14. En ella es pot veure com alguns camps són compartits per tots els events de que s'emmagatzemen, mentre que d'altres són particulars de cada event. Val a dir que l'estructura original allotja més camps ja que quan es comença el procés final de bolcat a disc de tota la traça final, els events del buffer de sistema es compatibilitzen amb aquesta estructura per tal de fer un procés de bolcat més senzill, però, donat que aquests camps addicionals no són usats pròpiament per al buffer d'usuari, no es comenten.

3.1.6 Creació d'una interfície nativa per als events d'usuari

Java ofereix una interfície estàndard de cara a permetre l'execució de codi natiu[19]: el JNI[20]. D'aquesta manera es permet la creació de passarel·les en el codi Java a seccions de codi que, per ser massa dependents de sistema o per ser-ne el rendiment un requisit indispensable, han de ser executades en codi compilat específicament per a una plataforma concreta.

Aquesta funcionalitat que presenta la JVM s'aprofita en JIS per tal de crear una plataforma de generació d'events des de codi Java. La finalitat és permetre als desenvolupadors generar events visualitzables des de Paraver des del codi Java de les seves aplicacions. L'avantatge d'això és poder introduir events en moments de l'execució que no tenen per què coincidir amb cap event de sistema.

---

[19] Codi compilat per a una arquitectura concreta i implementat en un llenguatge que no és Java (habitualment C)

[20] JNI o Java Native Interface

```
$javah -jni bjs.WebApps.ServletEvents
```

Figura 3.15: Generació d'un fitxer de capçalera de C a partir del codi Java

```
#ifdef __cplusplus
  extern "C" {
#endif
/*
 * Class: bjs_WebApps_ServletEvents
 * Method: GenericEvent
 * Signature: (JJ)V
*/
JNIEXPORT void JNICALL Java_bjs_WebApps_ServletEvents_GenericEvent
(JNIEnv *, jclass, jlong, jlong);
#ifdef __cplusplus
  }
#endif
```

Figura 3.16: Fitxer de capçalera generat a partir del codi Java per a la interfície JNI de JIS

La creació d'interfícies JNI consta de dues parts ben diferenciades:

- Creació d'una classe Java que servirà per a invocar els events des del codi Java
- Creació d'una implementació en C de les funcions a oferir

El procés comença amb la creació de la classe Java. En l'algorisme 11 es pot veure aquesta classe. Un cop creada, i sense oferir una implementació en java pels mètodes que conté, es realitza un preprocés de la classe per tal de generar un fitxer de capçalera de codi C per a les funcions a implementar. La comanda per a realitzar aquest procés es pot veure a la figura 3.15 . El resultat d'aquesta comanda és un fitxer de capçalera, com el mostrat, de manera abreujada, en la figura 3.16 .

Algorisme 11 Interfície Java per a la generació d'events des del codi Java

```
package bjs.WebApps;

public class ServletEvents {
  public static native void GenericEvent(long id, long val);
}
```

La implementació de les funcions de la classe de la interfície JNI s'han de realitzar dins d'una llibreria compartida que es carregarà en temps d'execució del codi Java. En el cas de JIS, i donat que tot el codi de l'instrumentador es troba implementat en una llibreria compartida, s'utilitza la pròpia llibreria de JIS per a implementar les crides natives des de Java. El codi implementat es pot veure en l'algorisme 12 . En el mateix es pot veure que l'única acció que es duu a terme és la inserció d'un event dins del buffer d'espai d'usuari a través d'una crida a una funció interna de JIS que realitza l'esmentada acció.

Pel que fa a la inserció d'events dins del codi Java, el procediment a seguir és el següent:

Algorisme 12 Implementació de la crida nativa per a la generació d'events des de codi Java

```
JNIEXPORT void JNICALL Java_bjs_WebApps_ServletEvents_GenericEvent
(JNIEnv *je, jclass jc, jlong id, jlong val)
{
  insert_servlet_generic_event((long)id,(long)val);
}
```

1. Importar el *package* implementat en el codi Java
2. Carregar la llibreria on es troben implementades les funcions natives
3. Invocar les funcions natives

El tres passos anteriorment esmentats han d'anar complementats amb la inserció en el *classpath*[21] del directori on es troba la classe Java creada. En l'algorisme 13 es poden observar els passos presentats.

Algorisme 13 Invocació de mètodes natius des de codi Java

```
import bjs.jis.JisEvents;
public class .... {
  public static void main (String args[]) {
    System.loadLibrary("jis");
    ...
    bjs.jis.JisEvents.GenericEvent(15000,1);
    ...
  }
}
```

La interfície nativa que JIS incorpora, fàcilment ampliable d'altra banda, és la que s'ha anat presentant al llarg d'aquest punt. Es tracta bàsicament d'un crida (*GenericEvent*) que serveix per a generar qualsevol event que es pugui necessitar. En Paraver els events consten d'un identificador i un valor. D'aquesta manera, els paràmetres que la funció implementada rep són, justament, dos nombres: el primer fa d'identificador i el segon de valor. D'aquesta manera es cobreix tot el ventall de possibilitats d'events que es vulguin generar des del codi Java de manera senzilla i fiable.

3.1.7 Consideracions sobre Time Stamping

Donat que el desenvolupament de JIS s'ha dut a terme sobre la plataforma Linux IA32, per tal d'aconseguir una màxima resolució en el temps es va decidir usar el comptadors hardware de cicles dels propis processadors per a extreure'n un *TimeStamp* de cada event. Per tant, la resolució de temps que s'ofereix en les traces Paraver generades amb JIS és de cicles de processador.

En concret, sobre els processadors Intel Pentium i superiors, existeix un registre anomenat *Time Stamp Counter* (TSC) que manté un comptatge hardware del nombre de cicles que han passat des de l'últim *reset* del processador, tant per un reinici de la màquina com per l'engegada inicial.

El registre és de 64 bits i es pot consultar a través d'una instrucció màquina que no requereix privilegis especials, anomenada *rdtsc* (*read time stamp counter*). Això comporta tres fets importants:

- Es pot obtenir el valor del *time stamp* tant des d'espai de sistema com des d'espai d'usuari amb una font de temps comuna i per tant síncrona.

---

[21] Variable d'entorn de sistema usada per la JVM per a trobar les classes a utilitzar (directament o referenciades dins del codi) en l'execució del codi Java

- Cal treballar amb valors de 64 bits en una arquitectura de 32 bits
- L'*overflow* del registre en una màquina moderna, amb un rellotge a 1 Ghz per exemple, és produeix a partir d'un ordre de temps superior a l'any, de manera que és un tema que, en principi, no hauria de suposar un problema.

El primer punt facilita la generació independent d'events des de dos espais diferenciats (usuari i sistema) ja que l'origen de temps és comú. El segon punt, com es detallarà a continuació, implica que s'han de tenir en compte certs detalls addicionals als habituals a l'hora de programar. El tercer és només una referència per tal de conferir fiabilitat al sistema.

Treball amb valors de 64 bits sobre una arquitectura de 32 bits

El suport per a treballar amb valors de 64 bits sobre una arquitectura de 32 bits l'ofereix el compilador utilitzat. En aquest cas, es tracta del gcc (un compilador de C habitual en les plataformes UNIX) de la GNU[10]. El suport es dóna a través de l'ús del tipus *long long*, i es tradueix en l'adaptació de les diferents operacions natives de 32 bits de la màquina per a estendre-les i poder treballar sobre 64 bits. Això comporta, per exemple, que una suma de dos valors de 64 bits no es tradueix com una suma en codi màquina sinó com la concatenació de dues sumes de 32 bits, una pels bits baixos i una pels alts, amb el corresponent tractament del *carry*.

El problema amb aquestes operacions va aparèixer a causa de la versió de gcc que es va utilitzar inicialment per al desenvolupament de JIS. Es tracta de gcc 2.96 que incorpora per defecte la distribució de Linux Red Hat en les seves versions 7.x, i que presenta nombrosos *bugs* ja que és considerada per la gent de GNU com una versió inestable.

Els *bugs* del compilador suposaven errors a l'hora de coordinar els resultats de les operacions de 32 bits natives de l'arquitectura quan es retornaven valors des de funcions, en determinades condicions. La primera solució va passar per obtenir el temps passant un paràmetre per referència a una funció enlloc d'esperar-ne un valor de retorn, tal com es veurà més endavant. Això va suposar la solució al problema, però de totes maneres es va canviar el compilador de desenvolupament per un gcc versió 2.95.3, l'última *release* realment estable feta per la gent de GNU. En l'algorisme 14 es pot veure la rutina utilitzada per a obtenir el temps de sistema en JIS (*gettime)* i la implementació en el kernel (*get_ cycles)* de la crida per llegir el rellotge hardware.

Algorisme 14 Funcions per a l'obtenció del temps a partir del comptador hardware

```
void gettime(unsigned long long *foo) {
  *foo = get_cycles();
}

static inline cycles_t get_cycles (void) {
#ifndef CONFIG_X86_TSC
  return 0;
#else
  unsigned long long ret;
  rdtscll(ret);
  return ret;
#endif
}
```

Temps en sistemes multiprocessadors

En sistemes multiprocessadors existeix una consideració especial a tenir en compte a l'hora d'obtenir el temps dels events. La qüestió radica en el fet que cada processador incorpora un comptador de cicles propi, de manera que aquests poden presentar un cert desplaçament els uns respecte als altres. I encara més tenint en compte que la especificació de sistemes multiprocessadors creada per Intel (Intel MP[5]) determina que un processador (el principal) serà el primer en arrancar en fer el *boot* de la màquina i que aquest serà l'encarregat d'inicialitzar la resta de CPUs. D'aquesta manera, sembla que el desplaçament entre rellotges es troba garantit.

Cercant en el codi Linux d'inicialització de sistemes multiprocessadors, per a les versions de *kernel* 2.4.x, es troba que un cop inicialitzats tots els processadors del sistema s'arriba a un punt en el codi on totes les CPUs entren en un bucle simultàniament. A cada volta d'aquest bucle, es fa un *reset* del comptador de cicles de CPU. D'aquesta manera es fa una aproximació força acurada a una sincronització de rellotges.

Per a provar l'efectivitat d'aquesta mesura, es va procedir a la creació d'un petit codi que senzillament creava 4 threads sobre una màquina Intel multiprocessadora de 4 vies i es mesurava (mitjançant un bucle infinit per cada thread i CPU) el desfasament de rellotges entre processadors. El resultat va mostrar que en cap cas el desfasament era superior a 200 cicles en una màquina funcionant amb una freqüència de rellotge de 700 Mhz, i per tant, sembla negligible.

3.1.8 Generació de la traça final i bolcats temporals

El procés final de la instrumentació de les aplicacions consisteix en la construcció i bolcat definitiu de la traça Paraver. Aquest procés es realitza un cop finalitzada l'aplicació, de manera que els recursos dedicats per a la seva consecució no afecten als resultats obtinguts durant la instrumentació de l'aplicació.

Previ al bolcat final, és possible que s'hagin dut a terme bolcats parcials dels buffers de JIS. En cas que aquests es trobin propers a emplenar-se, es procedeix a emmagatzemar la informació que contenen en un fitxer temporal i poder augmentar-ne la capacitat. Cal comentar aquí que el sistema de bolcats temporals no s'ha implementat igual per a l'espai d'usuari que per al de sistema, i tampoc l'èxit aconseguit en les dues versions ha estat equivalent.

Els bolcats temporals des d'espai d'usuari es produeixen quan, per cada event que s'hi vol intrioduir, es sol·licita una posició de memòria del buffer. Això es fa amb la rutina presentada en l'algorisme 15. En ella es veu com, quan el buffer es troba ple, es crida a una funció encarregada de bolcar el contingut de memòria en un fitxer temporal de disc. Com es veu, és un procediment molt senzill.

Algorisme 15 Bolcat temporal del buffer d'usuari

```
inline void *get_elem(void) {
  if(buffer.size == MAX_ELEMS) {
    dumpTempTrace(buffer.data, buffer.size,tmpDump);
    buffer.size = 0;
  }
  return &(buffer.data[buffer.size++]);
}
```

Amb el bolcat del buffer de sistema arriba un problema major: no es pot introduir retards ni bloquejos importants en la introducció d'events de kernel, ja que aquestes accions es duen a terme des de rutines crítiques de sistema (com la realitza els canvis de context entre *threads*). D'aquesta manera, s'havia de buscar un remei a aquesta situació. La opció escollida, sense gaire èxit tal com es veurà, consisteix en enviar un signal des del kernel a l'espai d'usuari quan l'ús del buffer supera

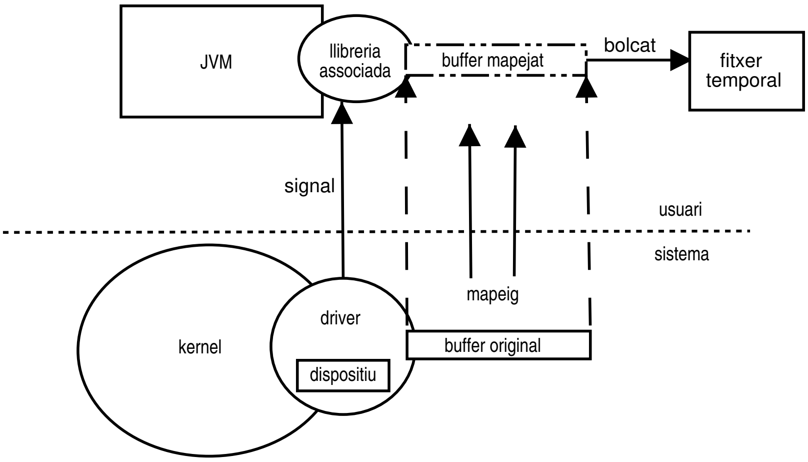

Figura 3.17: Esquema de bolcat a disc del buffer de sistema

el 50% de la seva capacitat, amb la idea que l'espai d'usuari, aprofitant que té el buffer de sistema mapejat, en faci un bolcat sense que les rutines de sistema se n'hagin de preocupar. Un esquema del que es planteja es pot veure a la figura 3.17. En l'algorisme 16 es pot veure la rutina encarregada de tractar l'esmentat signal.

Algorisme 16 Rutina d'atenció al signal de bolcat temporal del buffer de sistema

```
void buffer_signal(int iSig,
                   struct sigcontext_struct scs) {
  struct sigaction act;
  unsigned long i;
  dumpBufferTrace(threads, bufferDump);
  act.sa_handler = (void (*))buffer_signal;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  sigaction (SIGALRM, &act,NULL);
}
```

El problema amb aquesta solució basada en l'enviament d'un signal és que no garanteix un temps de resposta màxim. D'aquesta manera, durant l'experimentació feta amb aquest mètode, es va poder comprovar com el buffer de sistema s'omplia, durant el procés d'inicialització de Tomcat, molt abans que el signal fos rebut pel codi d'usuari implementat en la llibreria compartida de JIS. Per aquesta raó, i a falta d'una solució millor, es va optar per abandonar temporalment aquesta tècnica i optar per a redimensionar el buffer a una capacitat major que no requerís els bolcats temporals a disc.

4 Resultats i Aplicacions

4.1 Aplicacions de JIS

4.1.1 LU benchmark com a servlet (Linux)

Presentació

La LUAppl[3] és un *benchmark* habitual en plataformes paral·leles. El *benchmark* consisteix en el nucli d'una reducció LU completa realitzada sobre una matriu bidimensional d'elements de doble precisió.

En aquest cas, l'objectiu de les proves amb aquest *benchmark* no era valorar el rendiment de cap màquina ni de cap optimització sobre el sistema o el propi algorisme. Es pretenia estudiar quin és el comportament d'un servlet quan des de dins d'un servlet[1] es creen nous *threads.*

Aquesta prova pot ser representativa del comportament que pot presentar una aplicació executada sobre un servidor d'aplicacions, i és que aquests entorns es basen en continguts dinàmics i no estàtics, com els testats amb la modificació de la política de planificació de connexions de Tomcat, presentada anteriorment.

Els continguts dinàmics, a diferència dels estàtics, es caracteritzen de manera general perquè tenen un temps de procés important i una sortida cap al client relativament petita. En canvi, els continguts estàtics (servits des d'un servlet específic, com és el cas de Tomcat funcionant de manera autònoma) es caracteritzen perquè el temps de procés que presenten es limita a realitzar lectures del disc, i majoritàriament es dóna una quantitat de sortida cap al client força important (com a mínim en relació al temps de procés invertit).

La LUAppl, en versió servlet, es troba en l'extrem màxim de la situació presentada: com a servlet de contingut dinàmic que és, pot presentar un temps de càlcul molt gran (tant com es vulgui dimensionar la matriu que processa) i no produeix gairebé cap sortida cap al client (només un missatge de finalització i el temps en microsegons invertit en el càlcul).

Entorn de proves

De nou, la màquina escollida per a les proves és la monoprocessadora detallada en la figura **??**. El raonament és el mateix que el presentat en el punt **??**, ja que aquí tampoc s'intenta mesurar el rendiment de cap plataforma.

Modificacions en la LUAppl

Per tal de fer possible l'execució del benchmark basat en la reducció LU com a servlet, la implementació original d'aquest algorisme ha hagut de ser modificada per tal de donar resposta als requer-

---

[1] La particularitat d'incloure la creació de *threads* dins d'un servlet, és que els servlets per definició han de ser *thread-safe*, és a dir, que s'han de comportar de la mateixa manera si dins del seu codi hi ha 1 o més *threads* recorreguent-lo alhora. Això implica per exemple que, en cas de tenir dades compartides entre *threads*, els accessos exclusius els haurà de controlar el desenvolupador del servlet. Aquest fet fa que es presentin certs dubtes sobre quin comportament prendrà el sistema quan des de dins d'un codi *thread-safe* es creïn nous *threads*. En Tomcat, els *threads* que recorren els servlets són aquells que inicialment es creen com a *background threads* dels objectes HttpProcessor.

```
IMPORTACIONS ADDICIONALS
import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

DECLARACIÓ ORIGINAL
public class LUAppl

DECLARACIÓ MODIFICADA
public class LUAppl
       extends HttpServlet
       implements SingleThreadModel

COS PRINCIPAL ORIGINAL
public static void main (String args[]) {
  SIZE = 128;
  new LUAppl().go();
}

COS PRINCIPAL MODIFICAT
public void doGet(HttpServletRequest request,
                  HttpServletResponse response) {
  SIZE = 128;
  go();
}
```

Figura 4.1: Modificacions efectuades sobre el codi Java de la LUAppl per a poder utilitzar-la com a servlet

iments que eren imposats per l'execució dins d'un servidor d'aplicacions com és Tomcat. Els canvis més significatius aplicats es poden veure en la figura 4.1.

Principalment, s'hi pot veure que tres són les modificacions portades a terme:

1. Importació dels *packages* necessaris per a poder utilitzar la interfície estàndard dels servlets
2. Implementació del mètode doGet de la interfície comuna dels servlets
3. Revisió del codi per a fer-lo *thread-safe*

Visualització amb Paraver del comportament

La prova realitzada consisteix bàsicament en engegar el servidor Tomcat i invocar tres vegades consecutives el servlet que implementa la LUAppl. Aquesta acció dóna coma resultat una traça com la que es mostra en la figura 4.2. En ella es pot veure com les tres peticions són servides pel mateix objecte HttpProcessor (el número 4). Aquest *thread* és el que s'encarrega de dur a terme la tasca de *main* de la LUAppl, i és el creador dels altres dos *threads* que fan la computació real de la reducció LU (*Thread-0* i *Thread-2*). En la traça es pot observar que també hi apareix un altre *thread*, anomenat

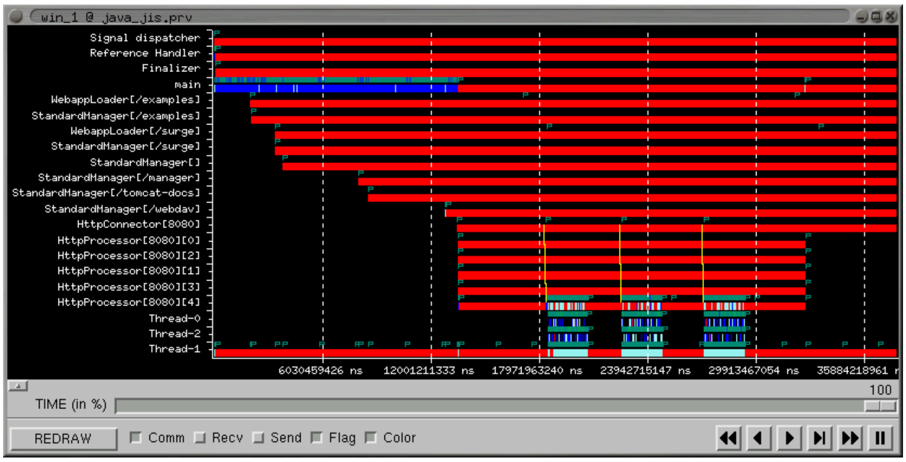

Figura 4.2: Aspecte general de l'execució de la LUAppl com a servlet 3 vegades consecutives

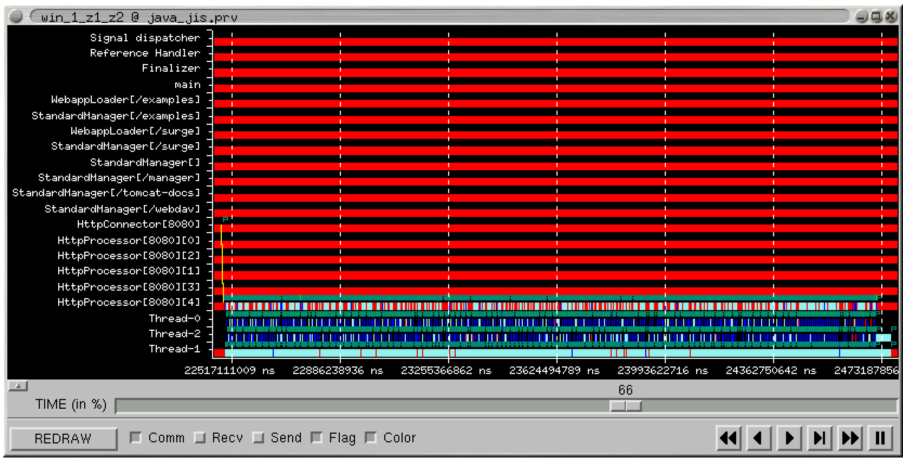

Figura 4.3: Aspecte general d'una execució de la LUAppl com a servlet

Thread-1 que no inicia la seva execució conjuntament amb els *threads* que realitzen els càlculs de la LUAppl. L'explicació de l'existència d'aquest *thread* l'hem de buscar en la implementació de *Pthreads* que es té sobre Linux i la seva forma de sincronitzar-se a través d'un *thread* extra, tal com s'explica a la pàgina a la pàgina 29. Com es pot veure, els *threads* creats dins del servlet funcionen de la manera que cabria esperar: es creen, realitzen un temps de computació (sense paral·lelisme perquè el sistema usat era un monoprocessador) i finalitzen. En aquell moment, el *thread main* també finalitza i l'execució del servei es dóna per finalitzada.

Per a veure amb una mica més de detall el procés d'execució d'una de les invocacions, cal veure la figura4.3. En aquesta figura es pot observar la gran quantitat d'activitat que realitza cadascun dels *threads* que intervenen en el procés de reducció de la matriu. En cap moment es dóna la situació que els dos *threads* es trobin corrent simultàniament, fet aquest que significaria algun error de programació en el codi de JIS.

Per tal de veure amb detall el procés d'assignació del *socket* des de l'objecte HttpConnector a l'objecte HttpProcessor es pot observar la figura 4.4. En ella es pot veure com l'event de comunicació de Paraver indica el pas d'activitat sobre el *socket* passa d'un *thread* a un altre. Després d'iniciar el procés sobre el *socket*, el *thread* realitza un cert temps de càlcul, després del qual, ja sap qui ha de servir la requesta rebuda. Aquest temps, visualitzat com una ràfega d'execució del *thread* de l'objecte HttpProcessor, és el que es necessita per a recórrer la jerarquia d'objectes que conformen el cor de

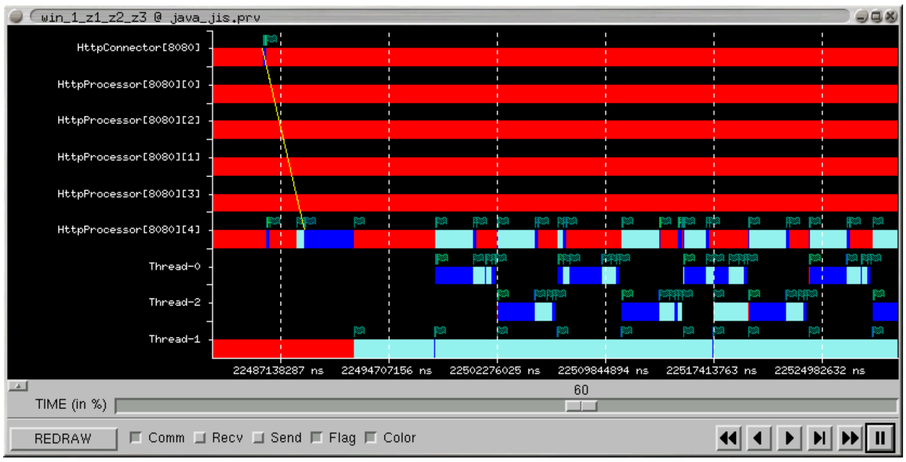

Figura 4.4: Detall de l'inici de l'execució de la LUAppl com a servlet

Tomcat abans d'arribar a la invocació del servlet LUAppl. Un cop iniciat aquest, es procedeix a la creació dels dos nous *threads* (quan l'espai horitzontal dedicat a un *thread* és de color negre indica que en aquell instant de temps no hi ha cap *thread* representat en aquell espai), que comencen a alternar els seus temps d'execució, conjuntament amb certes interaccions amb el *thread main*.

5 Conclusions

Amb aquesta implementació de JIS per a Linux IA32 s'ha verificat que les eines d'instrumentació multinivell (basades en el nivell de sistema i en el nivell d'usuari) són capaces d'oferir una informació molt completa sobre les aplicacions estudiades. Les aplicacions estudiades amb JIS queden exposades tant pel que fa a decisions preses pels desenvolupadors del software com per les accions internes del sistema operatiu, de manera que permet posar aquests dos elements en comú i veure com un afecta a l'altre i a l'inrevés.

El fet d'escollir Linux com a plataforma de desenvolupament permet, de manera clara, entrar amb molta més facilitat dins de les estructures del nucli del sistema i observar-hi què està succeint-hi amb els fluxos de l'aplicació.

En resum, doncs, JIS possibilita un estudi global de les aplicacions instrumentades que ofereix als desenvolupadors una nova perspectiva sobre el comportament de les seves aplicacions. D'aquesta manera, certs tipus de problemes relacionats amb la degradació del rendiment de les aplicacions Java que no tenen una explicació clara en el codi de les aplicacions poden ser examinats des del punt de vista del sistema operatiu, aspecte aquest no contemplat per la majoria d'eines d'instrumentació disponibles per a Java.

5.1 Treball Futur

Són diverses les línies de treball futur que queden obertes a partir del desenvolupament d'aquest projecte. A continuació se'n detallen algunes de possibles.

5.1.1 Ampliació de JIS - Linux amb nous events i suport al bolcat temporal del buffer de sistema

Amb la implementació actual de JIS per a Linux s'ha intentat cobrir totes les necessitats d'informació que es preveu que es poden necessitar inicialment per tal d'estudiar el comportament de servidors d'aplicacions basats en Java. De totes maneres, altres tipus d'events poden ser requerits de manera general o per algun tipus concret d'instrumentació que es vulgui realitzar. En aquests casos, l'ampliació de JIS per tal de cobrir aquestes noves necessitats és una possible feina a realitzar. També s'hauria de considerar la implementació d'un mètode de bolcat del buffer d'espai de sistema a disc en moments de saturació, donada la poc exitosa implementació feta en aquest projecte.

5.1.2 Ampliació de l'eina JIS per a donar suport a entorns SDSM

SDSM (Software Distributed Shared Memory) és un projecte desenvolupat dins del marc del CEPBA en què es pretén crear un espai d'adreces lògic comú entre diferents màquines d'un cluster de computadors. SDSM és l'encarregat de gestionar, entre d'altres coses, el moviment de pàgines de memòria entre els diferents nodes i, evidentment, la consistència general de l'espai tot l'espai d'adreces al llarg dels diferents nodes que conformen el cluster. Un pas endavant de JIS seria donar suport a aquest tipus de sistemes, i ser capaç d'instrumentar la JVM corrent-hi al damunt, oferint informació addicional relacionada amb la gestió realitzada per la capa SDSM.

Bibliografia

[1] J. Bartolomé and J.Guitart, A Survey on Java Profiling Tools. Research Report number: UPC-DAC-2001-13/UPC-CEPBA-2001-10

[2] J. Guitart, J. Torres, E. Ayguadé, J. Oliver and J. Labarta. Java Instrumentation Suite: Accurate Analysis of Java Threaded Applications. Proc. of the Second Annual Workshop on Java for High-Performance Computing, ICS00. St. Fe, New Mexico, May 7, 2000

[3] J. Oliver, E. Ayguadé and N. Navarro. Towards an efficient exploitation of loop-level parallelism in Java. In proceedings of the ACM Java Grande Conference, St. Francisco (CA), June 2000.

[4] A. Serra, N. Navarro, T. Cortés. DITools: Application-level support for Dynamic extension and flexible composition. In proceedings of the USENIX Annual Technical conference, June 2000

[5] Intel MultiProcessor Specification
http://www.intel.com/design/pro/datashts/242016.htm

[6] CEPBA (Centre Europeu de Paral·lelisme de Barcelona)
http://www.cepba.upc.es

[7] BJS (Barcelona Java Suite)
http://www.cepba.ucp.es/BJS

[8] Paraver
http://www.cepba.upc.es/tools/paraver

[9] *Java Virtual Machine Profiler Interface*
http://java.sun.com/j2se/1.3/docs/guide/jvmpi/jvmpi.html

[10] GNU (*GNU's Not Unix*)
http://www.gnu.org