UPC - E.T.S d'Enginyeria de Telecomunicació de Barcelona

PROJECTE FI DE CARRERA

PROJECTE FINAL DE CARRERA

Any: 2010

Alumne: Eva Isabel Soto Guiberteau

Títol del projecte: "Estudio e implementación de un sistema de obtención de trazas de entrada y salida y de reejecución de aplicaciones en función de las trazas obtenidas, para aplicar a sistemas de agentes móviles."

Director del projecte: Dr. Óscar Esparza Martín

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Acerca del proyecto

Este Proyecto Fin de Carrera ha sido desarrollado en el Information Security Group (ISG)[1] del departamento de Ingeniería Telemática (ENTEL)[2] de la Universidad Politécnica de Catalunya (UPC) [3], bajo la supervisión del Dr. Oscar Esparza Martín[4], profesor agregado de la Escola Técnica Superior d'Enginyeria de Telecomunicació de Barcelona (ETSETB)[5], miembro del ISG del departamento de Ingeniería Telemática de la UPC.

El tiempo invertido en el desarrollo del proyecto en total se divide en los siguientes temas:

- Estudio Previo: 20 %. Durante esta fase se ha invertido el tiempo en el estudio de la documentación sobre la tecnología de agentes móviles, sobretodo en el tema de la seguridad de los agentes. Lectura de las propuestas realizadas hasta la fecha para los problemas de seguridad resueltos y no resueltos, con sus pros y contras. Planificación de objetivos y acotación de áreas de búsqueda.

- Búsqueda de soluciones: 20 %. Durante esta fase se ha invertido el tiempo en la búsqueda de herramientas de obtención de trazas en código Java compilado. Comprobación de validez de las herramientas encontrada y estudio de las más indicadas para la resolución del problema propuesto. Selección y pruebas de una de las herramientas para la implementación de las soluciones (Javassist).

- Desarrollo de la solución: 30 %. Durante esta fase se ha invertido el tiempo en el desarrollo de posibles soluciones al problema propuesto. Durante la

[1] ISG Website http://isg.upc.es
[2] ENTEL Website http://www-entel.upc.es
[3] UPC Website http://www.upc.es
[4] Website http://sertel.upc.es/oesparza
[5] Website http://www.estseb.upc.es

misma se han hecho las pruebas y ajustes necesarios para la obtención de unos resultados satisfactorios.

- Pruebas y análisis de resultados: 10 %. Durante esta fase se han realizado las pruebas en el código desarrollado y se han analizado los resultados obtenidos para verificar la validez o no validez de las soluciones propuestas. Entre esta fase y la anterior se han obtenido los pros y contras de las soluciones encontradas en función a la dificultad de desarrollo, introducción de retardos en la ejecución o reducción del rendimiento de las aplicaciones.

- Documentación: 20 %. Durante esta fase se ha recopilado la documentación generada durante el resto de fases del proyecto y se ha desarrollado tanto la memoria final como los documentos necesarios para la presentación del proyecto ante el tribunal.

1.2. Motivación y Objetivos

La sociedad actual esta marcada por una clara necesidad de comunicación. Comunicación no solo entre seres, sino también entre entes de todo tipo de naturaleza. Con esto nos referimos al hecho de que la estructura del paradigma de acceso a la información, se ve acechada por todo tipo de accesos. Hoy en día coexisten usuarios humanos con usuarios virtuales, en un mundo lleno de información que debe ser accesible en un tiempo razonable, y con una calidad de servicio que debe ofrecer unos requisitos mínimos.

Como ya sabemos, una cosa es el servicio, y otra muy diferente el sistema. Como sistema de información entendemos todo aquello que hace posible la oferta de un servicio, pero que se intenta que quede lo más transparente posible al usuario final. Por servicio se entiende la evolución de lo que una vez fue necesidad, pero que actualmente pasa a la fase de poder representar un hecho.

Con todo esto, cabe destacar que el marco en el que se mueve este Proyecto Fin de Carrera, es el marco del sistema. Así pues, gracias a la tecnología que se está estudiando, se pretende poder ofrecer una solución funcional a una idea alternativa que al mismo tiempo es innovadora en lo que a movilidad se refiere, al ya conocido mundo de los sistemas distribuidos de información. El Proyecto se centra en una tecnología que funda sus raíces en el año 1996, en el laboratorio de desarrollo de la IBM en Tokio, cuyo grupo de desarrollo creó el denominado Aglets Team. Esta tecnología recibe el nombre de Agentes Móviles, y en concreto, de las diferentes opciones de agentes móviles a escoger, se usa una plataforma llamada Aglets Workbench, cuyo lenguaje de programación es Java. Este lenguaje permite la portabilidad del código entre diferentes plataformas, gracias a la interpretación del mismo llevada a cabo por las diferentes Máquinas Virtuales programadas específicamente para cada plataforma.

Un agente móvil es una entidad software formada por un código, unos datos y un estado, que puede migrar de forma autónoma de una máquina a otra ejecutando su código.

Un sistema de agentes no es más que una serie de servidores (en adelante serán llamados hosts) que soportan la ejecución de agentes. Los hosts están conectados mediante una red de manera que los agentes se pueden desplazar de uno a otro (migrar). Los agentes móviles no están limitados al host donde se inició su ejecución, si no que son capaces de viajar a través de la red y ejecutarse en una o varias máquinas remotas. Un agente, una vez finalizada su ejecución, devolverá los resultados, si los tuviera, al host origen. Los agentes móviles constituyen una herramienta de gran interés para la mayoría de aplicaciones que requieren un alto grado de automatización. Un agente móvil puede buscar información en beneficio de su propietario, negociar y cerrar tratos en su nombre. Respecto a su movilidad, un agente móvil tiene capacidad para decidir a qué servidores moverse, para parar su ejecución, migrar a otro nodo (preservando su estado) y continuar su ejecución.

Una vez introducida la noción de movilidad de código, y asociando el sistema a la necesidad de ofrecer seguridad a los servicios que se quieran ofrecer bajo esta tecnología, surgen puntos críticos que frenan el desarrollo de este tipo de sistemas. Se consideran dos entidades implicadas en este tipo de sistemas: el agente y la plataforma de ejecución. Por tanto se esperan dos tipos de ataques básicos:

- Ataque a la plataforma de ejecución: Estos serian ataques del agente al host donde se ejecuta. En este caso, el agente móvil de forma maliciosa intenta aprovecharse de las debilidades de la plataforma de ejecución. Como pueden recibirse agentes de diversas procedencias, nadie asegura las buenas intenciones del código. Pueden tenerse mecanismos de autenticación que aseguren y verifiquen el origen del agente, pero aun y así, no es posible que el agente oculte, por ejemplo, virus o caballos de troya.

- Ataque al agente: Estos serian ataques del host al agente que ejecuta. En este caso, la plataforma de ejecución puede comportarse de forma maliciosa. Al viajar por la red migrando de host en host, el agente puede visitar hosts con varios niveles de confianza, o incluso algún host que pertenezca a una entidad hostil. El problema es más difícil de resolver que el punto anterior, ya que el host ejecutor tiene control total sobre el agente y el entorno de ejecución: los datos que transporta, el código, el modo de ejecución, las comunicaciones con el exterior y los resultados. En este caso, los ataques tienen la desventaja de la clara ventaja computacional de la plataforma.

Con todo esto surge la necesidad inminente de reforzar el sistema. Para el caso de la posible agresión por parte de un agente móvil, se ha creado un entorno de ejecución que protege en bastante medida a la plataforma, restringiendo el acceso a recursos del sistema. Pero para el caso de una agresión por parte de un host malicioso, no existe ninguna solución global que evite este posible tipo de acciones. Para solucionar este problema se han descrito e implementado diferentes soluciones, tanto de prevención de ataques como de comprobación

posterior, aunque aun no hay una solución estándar y eficaz que permita la evolución y el desarrollo de los sistemas de agentes móviles.

Una de las soluciones descritas se basa en un sistema de trazas criptográficas [Vig98]. En este sistema se quiere obligar a los agentes que se ejecutan por los diferentes hosts a almacenar un conjunto de trazas con sus entradas y salidas para poder comprobar, mediante la reejecución del agente con los datos de las trazas almacenadas, que un host, del que se sospecha, ha ejecutado el agente de forma correcta. Este sistema es de comprobación, después de la ejecución, y no de prevención de ataques a los agentes. Los ataques se producirían igualmente y, el emisor del agente, debería decidir de que host se fía y de que host no para pedir el fichero de trazas y poder reejecutar el agente para comprobar si realmente ha habido ataque o no. En el documento se describe que datos se han de almacenar en el fichero de trazas y cual va a ser el protocolo de petición y envío de los ficheros generados en las ejecuciones.

El objetivo principal de este Proyecto Fin de Carrera ha sido la de encontrar una manera de obtener las trazas descritas en el documento [Vig98] y la de reejecutar un código basándose en los datos almacenados en las trazas. Inicialmente el objetivo se basó en comprobar si de alguna manera ya se había implementado y, además, estaba disponible para ser usado. En ese caso se hubiese cogido el desarrollo ya hecho y se hubiese probado para comprobar la conveniencia de usarlo en sistemas de agentes móviles o no. Posteriormente el objetivo pasó a ser el de encontrar una forma de hacer la implementación para poder usarla y finalmente el de desarrollar, con las utilidades encontradas, y probar el funcionamiento del software necesario, tanto para la obtención de las trazas como la reejecución una vez obtenidas. Visto lo amplio del proyecto, en cuanto a plataformas, sistemas de programación, etc... se fueron definiendo un conjunto de supuestos que se han aplicado al desarrollo realizado.

1.3. Organización del Proyecto

La organización del proyecto es la siguiente:

Capítulo 1: Introducción En este capítulo se introduce el paradigma de los agentes móviles dentro de los sistemas distribuidos, así como los principales inconvenientes que presenta relativos a la seguridad. Se describe el problema de los hosts maliciosos, así como las principales propuestas de protección de agentes y de detección de ataques publicadas hasta el momento.

Capítulo 2:Trazas Criptográficas En este capítulo se describe la propuesta de detección de ataques a agentes por parte de hosts a investigar y desarrollar. Se introducen los primeros requisitos del sistema a desarrollar.

Capítulo 3: Conceptos Previos En este capítulo se describen los conceptos necesarios para entender el desarrollo realizado. Se describen las características

del lenguaje de programación tenidas en cuenta en el desarrollo, una vez seleccionado el lenguaje a utilizar como requisito de la implementación.

Capítulo 4: Soluciones propuestas En este capítulo se detallan las posibles soluciones encontradas en la investigación previa al desarrollo para la resolución e implementación del sistema de detección. Se proponen distintas soluciones y se selecciona una de ellas para realizar la implementación definitiva.

Capítulo 5: Desarrollo e implementación En este capítulo se detallan, por un lado, las herramientas utilizadas en el desarrollo y, por otro, las soluciones a implementar en cada uno de los casos propuestos. Se divide el desarrollo en dos partes diferenciadas: la obtención de trazas de la ejecución del código y la reejecución del mismo código en función de las trazas obtenidas anteriormente. Para cada una de las partes se detalla las acciones realizadas para obtener la implementación.

Capítulo 6: Conclusiones y líneas futuras En este capítulo se detallan las conclusiones obtenidas del desarrollo realizado y de los resultados obtenidos. También se detallan las siguientes tareas a realizar para obtener un sistema válido.

Capítulo 7: Anexo 1: Pruebas En este capítulo se detallan las pruebas realizadas para validar los desarrollos que se han ido realizando. Se detallan los datos de prueba usados en cada uno de los casos y los resultados de cada una de las pruebas.

1.4. Introducción a los sistemas distribuidos

El concepto de sistema distribuido se opone al de sistema centralizado que se basa en la existencia de una sola CPU, una sola memoria y un solo disco con uno o más puestos de trabajo. Esta arquitectura, la de sistema centralizado, era la que existía originalmente en la mayoría de compañías. Se trataba de un supercomputador que aceptaba peticiones de uno o más puestos de trabajo. Toda la inteligencia del sistema radicaba en el supercomputador, mientras que los puestos de trabajo solamente le hacían peticiones y esperaban respuesta. Estos supercomputadores eran caros, grandes y sofisticados. Además requerían que los operarios que hacían el mantenimiento tuviesen un alto grado de conocimiento, lo cual encarecía aun más el mantenimiento de los mismos. Cuando el supercomputador caía, todo el trabajo quedaba parado. Además para aumentar la potencia, se requería comprar un procesador mayor, dejando al antiguo inoperativo. La comunicación entre supercomputadores era casi inexistente, debido a que las redes no estaban muy evolucionadas.

Esta situación inicial de trabajo cambió debido al abaratamiento del precio de los procesadores, lo que permitió construir máquinas más potentes con

menor precio, y la aparición de Redes de Área Local (LAN), lo que permitió la interconexión entre estas máquinas, de forma que se pudiese compartir la información de forma fácil y coherente y, sobretodo, los recursos costosos. Con esto nacieron los primeros sistemas distribuidos.

Podemos definir un sistema distribuido como aquel en el que los componentes hardware y software están localizados en computadoras independientes, unidas mediante una red que permite la comunicación entre los componentes y que es vista por el usuario del sistema como un componente único, independientemente de la localización geográfica de los componentes del sistema. Además se pretende compartir entre los usuarios los recursos costosos del sistema y la información.

La arquitectura de la mayoría de aplicaciones existentes en los sistemas distribuidos es la que se conoce como arquitectura clientes-servidor. Los clientes son procesos que demandan servicios al servidor por medio de mensajes y esperan el resultado. Por su parte, los servidores son procesos que ofrecen servicios y que están a la espera de recibir peticiones por parte de los clientes para enviarles los resultados.

Uno de los factores más importantes en el diseño de aplicaciones es la forma de comunicar los procesos. De la forma seleccionada dependerá el rendimiento del sistema y aspectos relativos a la flexibilidad. Existen diferentes tipos de formas de comunicación:

- Intercambio de datos: en este caso solamente se intercambian datos, mientras que la lógica de la aplicación permanece estática en la plataforma de ejecución. Como ejemplo de este tipo de intercambio encontramos los mecanismos de traspaso de mensajes y la Invocación de Procedimientos Remotos (Remote Procedure Call o RPC)

- Intercambio de código: en este caso no solamente se intercambia código, sino que la la lógica de la aplicación también va migrando de plataforma de ejecución en plataforma de ejecución. Este tipo también se conoce como código móvil. Ejemplos de esta tipología pueden ser el código bajo demanda (Code On Demand o COD) o los agentes móviles (Mobile Agents o MA).

1.5. Agentes móviles

1.5.1. Definición

En sistemas distribuidos un agente se define, desde la perspectiva del usuario final, como un programa que le ayuda y que actúa en su nombre. Es decir, para los usuarios, un agente permite que se delegue trabajo en el. Desde la perspectiva del sistema, un agente es un objeto de software que está situado dentro de un entorno de ejecución y que es capaz de actuar de acuerdo a los cambios que se produzcan dentro del entorno. Además es autónomo, es decir, tiene control sobre sus propias acciones, es pro-activo y esta continuamente en ejecución. Adicionalmente, y de forma opcional, puede comunicarse con otros agentes,

puede viajar de un host a otro, se adapta en función de su propia experiencia previa y parece creíble para el usuario. Una de las características obligatorias de los agentes no es la inteligencia, característica que si es obligatoria para los agentes inteligentes. Los agentes inteligentes tienen la capacidad de ofrecer comportamientos tales como razonamiento, planificación, aprendizaje, etc...Sin embargo, para los agentes inteligentes la movilidad no es un atributo obligatorio, lo que si es obligatorio para los agentes móviles.

Un agente móvil se compone de las siguientes partes:

- Estado: por estado entendemos los valores de los atributos del agente en un instante de tiempo. Será necesario conocer el estado de un agente en el momento de la migración (a otro host) para poder reanudar la ejecución en el host al que se ha migrado en el mismo punto en el que se paró la ejecución en el host anterior.

- Implementación: por implementación entendemos el código que hace las funciones para las que esta diseñado el agente. Esta será necesaria para que el agente tenga una ejecución independiente del lugar donde esta se realice. Si no se tiene el código del agente en el momento de la ejecución será necesario algún tipo de conexión con el propietario del agente para obtenerlo.

- Intefaz: por interfaz entendemos las funcionalidades que permite el agente y las opciones de comunicación que tiene con otros agentes dentro del sistema. Será necesaria para poder comunicarse con el.

- Identificador: por identificador entendemos un código único dentro del sistema de agentes que permite reconocer un agente en un momento dado. Será necesario para reconocer y controlar las acciones del agente en cada momento.

- Propietarios: por propietarios entendemos tanto los desarrolladores del código como la persona en nombre de la cual negocia dentro del sistema. Estos serán necesarios para saber quienes tienen la responsabilidad tanto legal como moral de los agentes que se ejecutan dentro de un sistema.

Un agente móvil no está ligado al sistema donde se inicia su ejecución. Es creado en un entorno de ejecución, pero puede viajar libremente entre los hosts en la red, transportando su código (implementación) y estado con él hacia otro entorno donde reanuda la ejecución. Son la extensión del modelo cliente-servidor, donde una parte del código es enviada a uno o más servidores para ser ejecutado.

Se pueden definir dos tipos básicos de movilidad para los agentes, en función de los elementos que llevan consigo al migrar entre los hosts de la red del sistema:

- Movilidad débil: es aquella en la que el agente no lleva la información de su estado al migrar entre los hosts de la red donde va ejecutándose. El código que se ejecuta es siempre el código completo, siempre se parte desde el mismo estado inicial.

- Movilidad fuerte: es aquella en la que el agente se lleva la información de su estado consigo al migrar entre los hosts de la red donde va ejecutándose. El agente es capaz de detener la ejecución en un punto cualquiera de su código, obtener los datos de su estado y empaquetarlos, migrar al siguiente host del itinerario llevando su código y estado y allí desempaquetar los datos del estado y reanudar la ejecución en función del punto en el que quedo en el host anterior. Este punto de inicio de ejecución vendrá marcado por los datos del estado que lleva incorporado al migrar de host y será diferente en cada uno de los hosts del itinerario del agente.

De los dos tipos, el más sencillo de implementar es el agente que tiene movilidad de tipo débil, ya que al migrar solo necesita llevar consigo el código a ejecutar. Aún y así, para que esto sea sencillo se deben usar lenguajes de programación portables, como por ejemplo Java. La movilidad fuerte es más complicada de implementar, ya que, a parte de la dificultad de llevarse el estado del punto en el que finaliza la ejecución y obtener el mismo estado en el punto de reanudación una vez migrado el agente a otro host, la ejecución puede depender de recursos locales del host inicial, como bases de datos, ficheros de datos o dispositivos, que pueden no estar disponibles en el host destino una vez se reanude la ejecución.

El itinerario de un agente es el listado de hosts por los que migra y donde va ejecutándose, ya se trate de un agente con movilidad débil o fuerte. Este itinerario puede estar predeterminado por el usuario emisor (que genera y lanza el agente) o puede ir modificándose en función del entorno, por ejemplo evitando máquinas caídas o a las que no se puede acceder a través de la red y volviendo luego, cuando estén disponibles, para realizar la ejecución.

Además de moverse entre los hosts de la red, un agente tiene capacidad de comunicarse con otros agentes, tarea que se lleva a cabo a través de paso de mensajes. Otra de las características es que puede clonarse y migrar a otras máquinas.

La ejecución del agente puede comportar la obtención de un resultado. Si este fuese el caso, el agente puede decidir volver al host origen, donde entregar el resultado y finalizar la ejecución o enviar este resultado mediante un mensaje, sin la necesidad de volver al host origen de la ejecución.

1.5.2. Ventajas de los agentes móviles

La mayor parte de las ventajas que presenta el uso de los sistemas de agentes móviles están ligadas al rendimiento de las máquinas o de la red. Otras ventajas vienen dadas por el aumento de la flexibilidad de las aplicaciones de estos sistemas. Algunas de las ventajas más importantes son las siguientes:

Reducción de la carga de la red Actualmente la mayoría de aplicaciones se basa en el envío de peticiones y la espera de respuestas (tanto en las comunicaciones síncronas como asíncronas). Con los sistemas de agentes móviles se evitan estos envíos de mensajes por la red, por tanto ésta lleva menos mensajes de comunicación entre procesos cliente y servidor (al estar el proceso en

la misma máquina en la que están los datos). Esto implica una reducción en las saturaciones de la red, aumentando su eficiencia. También se reduce el tráfico en la red al dejar de intercambiarse información intermedia durante la ejecución (si un agente obtiene los datos de la bbdd directamente en el origen y solamente envía el resultado, por ejemplo).

Mejora de la latencia de la red El hecho de enviar los agentes a ejecutarse a máquinas remotas evita que se deba disponer de una conexión estable y rápida de forma continua, ya que el host de ejecución puede estar desconectado durante la ejecución del agente y conectarse para migrar a otro host o para enviar el resultado. Es por esto que estos sistemas están indicados para redes inestables y de gran latencia.

La latencia de la red mejorará con los agentes, ya que se evitan muchos mensajes de petición y de envío de resultados durante la ejecución (resultados intermedios que no son los que se buscan realmente y que pueden hacer saturar la red), de esta forma se puede usar la red para aplicaciones que necesitan respuestas rápidas y en tiempo real, enviando un agente a los sistemas que controlen directamente las aplicaciones sin tener que preocuparse de la disponibilidad.

Encapsulación de protocolos Hasta ahora toda la funcionalidad de las aplicaciones esta en los servidores, a los que los clientes hacen peticiones y esperan respuestas. Esto implica que, tanto el cliente como el servidor, acuerden de antemano cuales van a ser los procedimientos para obtener los resultados requeridos. Si los protocolos de comunicación evolucionan o cambian, los procedimientos de las peticiones y obtención de respuestas deben ser revisados para que continúen funcionando correctamente. En el caso de agentes, en ellos se define la funcionalidad a ejecutar, por tanto no se han de poner de acuerdo entre clientes y host para poder ejecutarlos.

Ejecución asíncrona y autónoma Hasta ahora, la mayoría de aplicaciones de sistemas distribuidos se basa en llamadas síncronas, donde el cliente realiza la petición y espera a que el servidor le devuelva el resultado que ha pedido. Esto obliga a que los dos estén siempre conectados entre si, si alguna de las dos partes pierde la comunicación también muere el proceso y no pueden obtenerse los resultados deseados. Con los sistemas de agentes móviles, el cliente lanza el agente y se despreocupa de el hasta que este regresa o le envía el resultado de la ejecución. De esta forma, tanto el cliente como el host ejecutor pueden desconectarse sin miedo a perder la ejecución del proceso o continuar haciendo otras tareas mientras se obtienen los resultados.

Es el agente se encarga de controlar la ejecución, siendo totalmente autónomo, y por tanto el cliente es libre de seguir ejecutando otros procesos si lo desea. Esto también mejora los tiempos de ejecución de los procesos, ya que se puede emplear el tiempo que antes se esperaba (en las peticiones asíncronas) para realizar otras tareas. El tiempo de proceso también se reduce al ejecutar las aplicaciones directamente donde están los orígenes de datos.

Los agentes son programados para tomar decisiones de forma autónoma. El usuario que lo lanza delega todo el trabajo en el agente y este es capaz de realizar acciones en su nombre, sin que el usuario emisor intervenga en la ejecución.

Adaptación dinámica y naturaleza heterogénea En los sistemas actuales, el servidor debe instalar, antes de recibir las peticiones, todo el software necesario para que la aplicación funcione. Con los sistemas de agentes móviles, el cliente puede extraer la funcionalidad del servidor y enviarla a los hosts donde ha de ejecutarse, donde se instalará de forma dinámica, sin necesidad de preacuerdos sobre los formatos de intercambio de datos ni la previa instalación del software de la aplicación. Esto añade flexibilidad a las aplicaciones de estos sistemas. Es el cliente el que define sus procesos en función de los resultados que busca y no debe decidir entre los servicios que ofrecen otros cual le es mejor para los resultados que quiere obtener.

Robustez y alta tolerancia a fallos El sistema se vuelve más robusto al dejar de depender de la disponibilidad de la red y de los sistemas. El agente puede moverse libremente por la red siguiendo su itinerario de hosts y, si alguna máquina por la que debe pasar esta caída o no es accesible, puede evitarla y continuar su itinerario por las máquinas que si están disponibles. El agente es capaz de volver más tarde al host o hosts incomunicados o caídos (una vez recuperadas las máquinas o la conectividad) para ejecutarse y así finalizar las tareas para las que ha sido diseñado y programado.

Los sistemas basados en agentes también pueden distribuir la carga de forma más eficiente, distribuyendo los agentes a ejecutar entre las máquinas en función de la capacidad de cada uno de ellos.

Los sistemas existentes ya contemplan algunas de estas características, por lo tanto hay que estudiar el sistema que se quiere implementar y en función de los objetivos de la aplicación (reducción del tiempo de ejecución, sistemas robustos, etc...) es posible que exista otra arquitectura más conveniente para el desarrollo de una aplicación que el uso de agentes móviles. Los sistemas de agentes móviles tienen muchas ventajas sobre los sistemas existentes, pero también tienen desventajas. En algunos casos las desventajas que comporta el uso de agentes móviles puede hacer reconsiderar la arquitectura de la solución a implementar en una aplicación. En el capítulo 1.5.3 se listan las desventajas principales del uso de agentes móviles.

1.5.3. Inconvenientes de los agentes móviles

Viendo la lista de ventajas derivadas del uso de agentes móviles parece, a simple vista, que sea la mejor opción en el desarrollo de aplicaciones. Sin embargo, el uso de agentes móviles también implica una serie de desventajas que en algunos casos puede llevar a desechar una arquitectura de este tipo para la implementación de una solución. Las desventajas principales que tiene el uso de agentes móviles son las siguientes:

Problemas de compatibilidad entre plataformas Los hosts por los que va migrando un agente y los recursos que se utilizan deben ser compatibles entre ellos para que la ejecución pueda llevarse a cabo. Una posibilidad de superar esta desventaja es usar lenguajes de programación interpretados como Java. Sin embargo, para poder ejecutar este tipo de código es necesario que los hosts tengan instalada la máquina virtual necesaria para llevar a cabo la ejecución. Para el uso de otros recursos como bases de datos o ficheros no es tan difícil, ya que existen soluciones estándares una vez seleccionado el lenguaje de programación (por ejemplo el lenguaje de consulta de bases de datos, SQL, es implementado por todos los fabricantes de sistemas gestores de bases de datos por igual, al menos una parte común).

Problemas derivados de la seguridad Debido a que el agente es enviado a la red para ir de host en host ejecutándose puede sufrir diversos ataques durante su ciclo de vida. El agente puede ser atacado mientras migra en la red (lo que se puede solucionar encriptando tanto los datos como el código) pero puede ser también atacado por el host que lo ejecuta. Una vez llega al host, este dispone del código, datos y control de ejecución del agente y el agente puede sufrir diferentes ataques por su parte. Puede leer datos que pueden ser confidenciales, negar la ejecución, modificar su código para obtener resultados que le beneficien, etc...Este problema se conoce como el problema de los hosts maliciosos y es de difícil resolución. A parte de los ataques que pueda sufrir un agente, también se puede dar el caso contrario, es decir, que sea el agente el que ataque al host en el que se ejecuta.

Las desventajas que comporta el uso de agentes móviles son de difícil resolución, sobretodo los temas relacionados con la seguridad, si es que el sistema a implementar requiere un nivel de seguridad alto. Se han hecho multitud de propuestas al respecto, pero no se ha encontrado aún una solución global que sirva para el despliegue masivo de las aplicaciones de este tipo.

1.5.4. Seguridad en Agentes Móviles

Con Internet, y en particular la Web, cada vez más popular y con el comercio electrónico empezando a despegar, la seguridad parece más importante que nuca. Con la seguridad adecuada, la gente no dudará en comprar e intercambiar información confidencial vía Internet. Además, la seguridad es uno de los factores clave en el arranque de la tecnología de agentes móviles. Como otros tipos de código descargable y ejecutable, los agentes móviles son una amenaza potencial para un sistema (pueden atacarlo). Pero ellos también están expuestos a amenazas por los hosts (pueden ser atacados por ellos), una situación que no se da con los actuales sistemas de seguridad. En ausencia de un mecanismo de seguridad para el agente simple y comprensible, los usuarios no usarán agentes o aceptarán que ningún agente móvil visite sus máquinas, y no veremos un desarrollo masivo de agentes móviles.

Como se ha comentado en el capítulo 1.5.3, el uso de agentes móviles comporta múltiples amenazas de seguridad, algunas de las cuales aun no tienen una solución que permita el desarrollo de aplicaciones de este tipo de forma masiva.

Las amenazas de seguridad en agentes móviles son las siguientes:

- Protección del agente:
  - El host remoto amenaza al agente: El agente está de viaje y visita hosts en los que no se tiene confianza y que pueden intentar extraer información privada o modificar el agente para que sea de alguna manera nocivo. Los posibles ataques incluyen modificación del código del agente, ejecución ilegal y acceso ilegal.
  - El agente amenaza a otro agente: Un agente interactúa con un o de nuestros agentes en un intento de extraer información privada o confidencial y para interrumpir su ejecución. Un posible ataque sería acceso ilegal.
  - Terceras partes no autorizadas amenazan al agente: Una entidad maliciosa puede alterar los mensajes que se intercambian entre los hosts o taponar la comunicación entre ellos y revelar el contenido del agente. Posibles ataques incluyen alteración y escuchas no autorizadas (eavesdropping).
- Protección del host:
  - El agente que llega amenaza al host: Un agente que visita el servidor intenta acceder y corromper los ficheros privados del host o interrumpe al servidor. Posibles ataques incluirían acceso ilegal, hacerse pasar por el agente (masquerade), caballos de troya, denial of service y rechazo (repudiation).
  - Terceras partes no autorizadas amenazan al host: Una entidad maliciosa puede enviar un número elevado de agentes "spam" al servidor para ponerlo fuera de servicio. Posibles ataques incluyen denial of service y replay.
- Protección de la red:
  - Agentes que amenazan a la red: Un agente empieza a multiplicarse y a viajar de forma masiva en la red en un intento de inundar la red con agentes. Posibles ataques incluirían el denial of service.

En nuestro caso, se busca solucionar el problema de ataques del host ejecutor al agente. El problema es conocido como el problema del Host Malicioso. Este ha sido estudiado y se han propuesto diferentes soluciones sin obtener éxito.

1.5.5. Aplicaciones de los agentes móviles

Existen multitud de servicios que pueden resolverse de forma satisfactoria con el uso de la tecnología de agentes móviles. Por ejemplo, su flexibilidad y su bajo coste hacen que pueda ser la solución ideal en el caso de tareas que tengan un alto grado de automatización. En muchas de las aplicaciones existentes, el uso de agentes móviles representaría una mejora en alguno de los aspectos del servicio.

Antes de decidirse por el uso de esta tecnología se debe tener claro en que aspectos se puede obtener una mejora, por ejemplo en el rendimiento de la red o en la facilidad de uso. En función del número de saltos que realiza un agente entre los hosts de la red podemos clasificarlos en dos tipos:

- Single-hop agent: como su nombre indica, solo realiza un salto desde el host origen al host destino, donde se ejecuta. Después de finalizar la ejecución, el agente vuelve al host origen a entregar los resultados de la ejecución. Este tipo de agentes está especialmente indicado para tareas que requieren el tratamiento de un gran volumen de datos o las que precisan de mucho tiempo de ejecución. En estos casos, es mejor enviar de forma paralela un agente a cada uno de los hosts del sistema que realizar la ejecución en serie haciendo que el agente vaya migrando de uno en uno y ejecutando su código.

- Multi-hop agent: en este caso el agente da múltiples saltos por la red, es decir va migrando de host en host y en cada uno de ellos se ejecuta su código. En el caso de agentes con movilidad débil (los que siempre se ejecutan desde la misma situación inicial) este tipo es el más indicado para realizar tareas simples y repetitivas en múltiples máquinas.

En los siguientes servicios el uso de la tecnología de agentes móviles podría suponer una mejora en alguno o varios de aspectos como aumento del rendimiento de la red, de la velocidad de ejecución, ect:

- Búsquedas por la red: el agente es enviado por el cliente con unos criterios de búsqueda preestablecidos. Este busca entre los distintos hosts del itinerario marcado los componentes que cumplan los criterios que se le han indicado.

- Control de equipos remotos: el cliente puede enviar a un agente a un dispositivo remoto donde tomará el control y lo configurará con datos que puede llevar incorporados. Además de la configuración, se pueden enviar agentes que realicen tareas de monitorización del estado del dispositivo, enviando eventos que se produzcan o recopilando estadísticas del uso de los mismos.

- Discriminación de información: haciendo uso de la propiedad de movilidad del agente se puede enviar información a un conjunto de máquinas predefinidas.

- Procesado distribuido en paralelo: en el caso de tener problemas complejos, se puede enviar agentes que repartan el proceso entre distintas máquinas. La computación puede realizarse tanto en paralelo, enviando el agente a los distintos hosts de la red, como en serie, indicando un itinerario al agente que irá de host en host hasta obtener los resultados requeridos.

- Comercio electrónico: los agentes se pueden programar para que realicen acciones en nombre de un usuario. De esta forma, indicándole unos requisitos máximos y mínimos puede realizar negociaciones de precios e incluso compras y ventas de productos. De esta forma es más sencilla la comparación de precios y prestaciones de los artículos que se quieren adquirir.

No todos los usuarios finales tendrán ni la capacidad computacional ni los conocimientos para realizar el envío o la ejecución de un agente. En este caso, una posible solución seria delegar la faena a un servidor especializado. Este agente especializado funcionaría a modo de repositorio, desde donde se podrían descargar agentes preprogramados o configurables por el usuario que los quiere usar. Por otra parte, también es dudoso que el usuario dictamine de forma directa el itinerario del agente. Por norma general, la decisión vendrá determinada por una consulta a un servicio de directorio o de páginas amarillas.

1.6. Introducción al problema de los hosts maliciosos

El problema de los hosts maliciosos ha sido estudiado recientemente y es de más difícil solución respecto a la seguridad en sistemas de agentes móviles. Esto es debido a que los hosts de la red en la que se ejecuta el agente tienen distintos grados de confianza, por tanto es fácil esperar algún tipo de ataque de los que se tenga menor grado de confianza. Al aceptar un agente y ejecutarlo, el host adquiere control total sobre el agente. El host conoce el código del agente y los datos que transporta (incluso los que sean confidenciales, como passwords, etc...), controla el flujo de ejecución, las comunicaciones que tiene y conoce y puede modificar el itinerario que sigue el agente. Por esto, el host podría intentar atacar al agente de muchas formas, ejecutando el código de forma incorrecta, cambiando los datos del estado, modificando el flujo de ejecución, bloqueando las comunicaciones, o eliminando algún host del itinerario, etc...

La solución que tratamos en el proyecto se apoya en el uso de criptografía para el cifrado de los mensajes que se envían para enviar las trazas. Para poder usar mecanismos de este tipo es necesario tener un entorno de confianza, donde se puedan cifrar y firmar datos, y sobretodo almacenar claves secretas sin peligro a escuchas o modificaciones. Sin este entorno, es imposible este tipo de solución. El agente no puede almacenar passwords ya que esta podría ser leída o modificada por el host. Esta es una de las razones por la que aun no se ha encontrado una solución global al problema del ataque a agentes por parte de los hosts ejecutores.

1.7. Tipos de ataque

Hay muchas razones para que un host inicie un ataque contra un agente. Las razones se pueden agrupar en dos tipos: cuando se ataca para conseguir un beneficio o cuando se ataca sin querer obtener beneficio alguno. Algunas de las razones del primer tipo son obtener un beneficio económico, obtener una ejecución favorable (cuando se buscan productos entre diferentes marcas, por ejemplo) o dañar la reputación de una tercera entidad. De entre las del segundo tipo tenemos un ataque por diversión, ataques por destrozar al agente o por demostrar un alto conocimiento del tema. Que los ataques sean de un tipo o de otro depende del escenario en el que se realiza la ejecución. Por ejemplo no intentará el mismo tipo de ataque un servidor en una multinacional que un PC anónimo en internet. Del primero se pueden esperar ataques del primer tipo, para obtener algún tipo de beneficio económico. Del segundo es más posible tener un ataque del segundo tipo. Por estas razones, antes de implementar cualquier esquema de protección se debe realizar el estudio de quienes son los hosts potenciales a atacarnos y las razones por las cuales pueden hacerlo.

Hay dos tipos principales de ataques:

- **Ataques pasivos:** son aquellos que no modifican el agente y la información que este contiene. En este caso es difícil comprobar que el agente ha sido atacado, ya que nada se modifica. Son principalmente escuchas en las comunicaciones para comprobar que tipos de agentes y mensajes se mueven en un sistema.
- **Ataques activos:** son aquellos que modifican el agente o los datos que este contiene. En este caso es más sencillo comprobar que el agente ha sido atacado, ya que alguna de sus partes ha sido modificada.

Los principales ataques son los siguientes:

- Suplantación de identidad
- Degradación de Servicio (DoS)
- Acceso no autorizado
- Repudio
- Manipulación
- Confabulación

1.7.1. Suplantación de identidad

En este tipo de ataque, una entidad se hace pasar por otra para usurpar sus recursos, ganar sus permisos de acceso, acceder a información confidencial o simplemente para dañar la reputación de aquel al que suplanta. En este sentido, en un sistema de agentes, pueden darse los siguientes casos:

Un agente suplanta la identidad de otro En este caso un agente suplanta la identidad de otro, para, como se ha comentado tener su nivel de acceso a los distintos hosts o acceder a información confidencial. Además de suplantar la identidad de otro agente, este pude volverlo malicioso a su vez.

Una plataforma suplanta la identidad de otra En este caso es el host el que suplanta la identidad de otro, haciendo creer a los agentes que es otro que podría tener un grado más alto de confianza, para que el agente se confíe.

1.7.2. Denegación de Servicio (DoS)

Este tipo de ataque se hace para dejar fuera de servicio algún recurso o servicio de la red. De esta forma el recurso o servicio no podría ser usado por ningún otro agente en la red. Este tipo de ataque suele ser difícil de evitar, aunque es muy fácil detectarlos.

Otro tipo de ataque de Denegación de Servicio sería cuando el host, que tienen control total sobre el agente, bloquea sus comunicaciones o su ejecución para que no funcione de forma correcta. Un agente también podría hacer un ataque de este tipo a un host consumiendo, en su ejecución, todos los recursos disponibles de la máquina. De esta forma no podría ejecutarse ningún agente más en ella, quedando fuera de servicio.

1.7.3. Acceso no autorizado

En este tipo de ataque se intenta acceder a recursos a los cuales no se tiene permiso de acceso. Los accesos más comunes de este tipo son escucha (acceso a datos considerados como confidenciales) y la alteración (modificación de los datos). Como se ha comentado, el host ejecutor tiene acceso total al código y datos de los agentes que se ejecutan en el y por tanto puede tanto sacar información del agente como modificar su comportamiento para que el resultado le sea favorable. Por parte del agente, este puede intentar acceder a los orígenes de datos o ficheros del host a los que no tiene acceso. El enviar el agente cifrado no es solución para este tipo de ataque, ya que no se puede evitar que el host conserve una copia del código y datos del agente para hacer un análisis posterior a la ejecución.

1.7.4. Repudio

Un host realiza un ataque de repudio cuando niega una acción que si tuvo lugar. Es posible evitar este tipo de ataques e incluso buscar responsabilidades si la información intercambiada esta correctamente firmada por la entidad emisora.

1.7.5. Manipulación

En este tipo de ataque se modifican el código o datos del agente.

Es posible asegurar la confidencialidad, integridad y autenticidad del código, datos o resultados que provienen de otros hosts usando técnicas como el cifrado o la firma digital. Sin embargo, es difícil detectar o prevenir las manipulaciones realizadas en el agente durante la ejecución. El host puede manipular cualquiera de los contenidos del agente durante la ejecución.

1.7.6. Confabulación

En un sistema abierto como Internet, las relaciones de confianza son limitadas. Por ello se podría suponer que es difícil que un grupo de hosts confabulen contra un agente ya que los confabuladores deben estar en el itinerario del agente, y además deben confiar entre ellos. Los ataques de este tipo son difíciles de detectar o prevenir, de hecho, la mayoría de propuestas publicadas hasta el momento son vulnerables a confabulaciones. Sin embargo, existen técnicas de protección del itinerario [MB03] que pueden ser utilizadas para limitar los efectos de las confabulaciones.

1.8. Servicios de Seguridad

Para diseñar los mecanismos de protección adecuados debemos conocer cuales son los requisitos de seguridad que buscamos en un sistema de agentes móviles. Dado que la mayoría de técnicas de protección se basa en técnicas de cifrado y firma digital, asumiremos que disponemos de una Infraestructura de Clave Pública o PKI y que todos los hosts disponen de un certificado de identidad válido. En función de los requisitos que se buscan, se usarán unos servicios de seguridad u otros. A continuación se enumeran los principales requisitos y servicios de seguridad:

Autenticación La autenticación consiste en que todas las entidades implicadas en el sistema estén identificadas y puedan demostrar quienes son. Este servicio sería válido para evitar suplantaciones de identidad o ataques de repudio. Para asegurar que la información que transporta un agente está autenticada, se puede usar la firma criptográfica. La entidad emisora firmaría la información y siempre se sabría a quien pertenece o quien realiza las acciones programadas en el agente.

Integridad La integridad en un sistema de agentes móviles implica que los contenidos de los agentes (código, datos, estado, itinerario y resultados) no son manipulados sin control. Si una entidad en el sistema no está autorizada no debería poder modificar los datos del agente. La integridad en las comunicaciones (en la migración de los agentes entre hosts) es fácil de conseguir mediante técnicas criptográficas comunes como la firma digital. En cambio, asegurar la integridad de un agente mientras se está ejecutando en un host no es tan sencilla de conseguir, ya que el host tiene control total sobre el agente y los datos que este contiene.

Confidencialidad La confidencialidad en un sistema de agentes móviles consiste en que las entidades no autorizadas no tengan acceso a los datos del agente ni al código que define su comportamiento. Como en el caso anterior, es fácil asegurar la confidencialidad en las comunicaciones (durante la migración entre hosts) con técnicas criptográficas tales como el cifrado de datos. Sin embargo, es difícil conseguir confidencialidad total del agente con el host que lo ejecuta, porque este tiene control total sobre el código que ejecuta. No se puede prevenir que el host extraiga datos que son confidenciales mientras ejecuta el código.

Autorización Por autorización se entiende los derechos que tiene un agente para acceder a cierta información en función de sus responsables. Es la capacidad de un agente para acceder a un conjunto de información o a otro.

Auditoría Es la capacidad de anotar y poder revisar posteriormente las acciones que un agente o host han realizado durante una ejecución. Analizando las acciones posteriormente se pueden detectar distintos tipos de ataque.

1.9. Medidas de seguridad en sistemas de agentes móviles

En función del agente que se está ejecutando se aplicarán unas medidas u otras de seguridad. No todas las medidas a aplicar serán de prevención de ataques. Tendremos dos tipos de medidas:

Medidas de prevención de ataques En los casos en los que los beneficios a obtener atacando sean mayores a la sanción a imponer será aconsejable usar técnicas de prevención de ataques. Esto será evitar que un agente sea atacado y no finalizar la ejecución si este está siendo atacado por alguien, ya que se pierde mas (sobretodo en coste) que lo que se ganaría con la multa al atacante. Estos son los casos que aun no tienen una solución o propuesta de solución viable. Se han publicado algunas propuestas de solución pero aun no se han conseguido implementar o usar de forma masiva por los sistemas de agentes móviles.

Medidas de detección de ataques En esta categoría incluiremos aquellas propuestas que pretenden detectar los ataques de manipulación después de la ejecución del agente. El objetivo de este tipo de propuestas es disuadir a los hosts maliciosos de realizar ataques, ya que si son detectados podemos tratar de imponerles un castigo. Los esquemas de detección de ataques son más fáciles de implementar que los esquemas que tratan de prevenirlos. Sin embargo, padecen de una serie de inconvenientes que impiden su aplicación en todos los entornos. En primer lugar, es necesaria una TTP (Trusted Third Party o Tercera Parte de Confianza) que arbitre entre todas las partes y que pueda imponer sanciones cuando sea necesario. Además, si el beneficio obtenido al realizar el ataque es superior al castigo que pueda imponerse, sin duda estaremos incentivando

comportamientos maliciosos. En este último caso es cuando seria aconsejable usar medidas de prevención en lugar de medidas de detección.

1.9.1. Técnicas de protección de las entidades de los sistemas de agentes móviles

Las entidades básicas a proteger dentro de un sistema de agentes móviles son dos: el agente y la plataforma. Los posibles ataques que se pueden producir a un agente se basan en dos tipos, los ataques al agente cuando este migra por la red de host en host y los ataques de la plataforma a los agentes que esta ejecuta.

En el caso de la protección de los agentes que transitan por la red, se pueden aplicar mecanismos criptográficos usuales, como son el cifrado de la información y el firmado digital. En el caso de protección de los agentes durante la ejecución, el problema no está aún resuelto. En este caso nos encontramos con el problema del "host malicioso" para el que se continúa buscando una solución. Para la mayoría de ataques de seguridad contra la plataforma es posible reutilizar medidas conocidas ya usadas en sistemas distribuidos convencionales.

1.9.1.1. Protección de la plataforma

La mayoría de ataques contra la plataforma son evitables o detectables con técnicas comunes de seguridad como son un control de acceso adecuado (comprobación de credenciales y establecimiento de privilegios) y la limitación del entorno de ejecución de los agentes (operaciones restringidas y límites de consumo de recursos) con mecanismos de tipo sandboxing como los utilizados en Java.

Como primera medida, el código y los datos de un agente deben estar firmados por la entidad origen para asegurar su integridad. Del mismo modo, cada host debe firmar los resultados intermedios para asegurar que llegan de forma íntegra al host origen. Estas medidas no impiden la ejecución de un posible código malicioso, aunque si facilitan la posible búsqueda de responsabilidades.

1.9.1.2. Protección del Agente Móvil

Los posibles ataques sobre los agentes móviles se dan en dos entornos diferenciados: mientras el agente migra de host en host por la red y durante la ejecución en los hosts que este visita. En los capítulos anteriores hemos visto los diferentes tipos de ataques que pueden recibir.

Durante la migración del agente, este se puede proteger mediante técnicas criptográficas comunes, por ejemplo usando conexiones seguras (SSL). También mediante técnicas de cifrado o de firma del código se pueden evitar ataques de suplantación de identidad o de no repudio.

Durante la ejecución del agente, la protección depende en gran medida del host que este visite y el grado de confianza que se tenga sobre el. En este caso no hay una solución conocida que de protección global al agente frente a los ataques realizados por la plataforma. Nadie asegura que la plataforma ejecute

el código de forma correcta o que permita la migración del agente a otros hosts. Tampoco se puede ocultar nada en el agente a la plataforma, esta tiene control total sobre los datos y la ejecución. Los ataques de otros agentes maliciosos si que tiene fácil solución separando los dominios de ejecución de los dos agentes para evitar la interacción directa.

Capítulo 2

Trazas Criptográficas

2.1. Introducción

Este proyecto tiene como objetivo encontrar una solución a los problemas de seguridad que presenta el uso de agentes móviles. En concreto se busca encontrar soluciones al problema del host malicioso. Este problema se da cuando los host que forman parte del itinerario de un agente, y que tienen control total sobre la ejecución (código y datos) del agente, le atacan, manipulando el código o los datos para obtener algún tipo de beneficio con ello.

En 7.10 se introduce el mecanismo de obtención de trazas criptográficas como solución al problema del host malicioso. En el documento se describe el mecanismo de protección, a posteriori, basado en el almacenamiento de trazas para comprobar, en los casos en los que se sospeche de la ejecución, la correcta ejecución del agente en cada host. El documento indica tanto los datos a almacenar en las trazas como el protocolo de envío de trazas entre los hosts de ejecución y el host de envío original del agente. El protocolo de envío se basa en el uso de criptografía, de ahí el nombre del mecanismo: trazas criptográficas.

El almacenamiento de trazas criptográficas es un mecanismo de detección de manipulación ilegal (tampering) de la ejecución de un agente. Mediante este mecanismo el propietario del agente puede verificar con un alto grado de certeza si su agente ha sido ejecutado conforme a su especificación, es decir, su código. Con las trazas obtenidas es posible conocer si hay algún bloque de código por el que no se ha pasado en la ejecución, lo que haría sospechar que el host que lo ejecutó manipuló el agente.

El mecanismo presenta una serie de inconvenientes, el más importante de los cuales es que es un mecanismo de protección a posteriori. Esto quiere decir que permite la detección de comportamientos ilegales o maliciosos después de la ejecución del código del agente, con lo cual cualquier actuación maliciosa que se introdujese en la manipulación no se podría evitar, al ser siempre ejecutado el agente. Además de esto, el mecanismo no proporciona métodos para saber si la manipulación del código ha tenido lugar realmente, es necesario el análisis de

los resultados de la ejecución del agente para tener una sospecha de que el host ha manipulado al agente de algún modo.

2.2. Trazas Criptográficas para Agentes Móviles. Protocolo de envío.

Como se ha comentado en capítulos anteriores, un agente se compone de dos partes: un segmento de código y el estado. El segmento de código se compone de un conjunto de sentencias o instrucciones a ejecutar. Suponemos que este código es constante a lo largo de la vida del agente, es decir, que el agente no puede modificar su segmento de código como resultado de su ejecución. El estado incluye las estructuras globales de datos, la pila de llamadas y el contador del programa. Este estado, al contrario que el segmento de código, es dependiente del tiempo y de la ejecución del agente.

El segmento de código, como se ha comentado, se compone de sentencias o instrucciones a ejecutar para cumplir la funcionalidad del agente. Estas sentencias pueden ser de dos tipos:

- Sentencias blancas: estas sentencias son aquellas que cambian el estado del agente basándose solamente en valores de variables internas del programa que se está ejecutando.
- Sentencias negras: estas sentencias son aquellas que cambian el estado del agente basándose en valores provenientes del entorno externo de ejecución.

De estas sentencias se guardarán las trazas durante la ejecución. Una traza de ejecución *(T^p)* de un programa *(p)* se compone de una secuencia de pares <n,s> , donde:

- n representa un identificador único de la traza.
- s representa la firma de la sentencia ejecutada. Cuando se asocia con una sentencia negra, la firma contiene los valores nuevos que asumen las variables internas después de la ejecución de la sentencia. Cuando se asocia con una sentencia blanca la firma está en blanco.

Para poder aplicar de forma correcta este mecanismo suponemos que se cumplen las siguientes condiciones:

- Asumimos que todos los protagonistas involucrados poseen una clave pública y una privada que pueden usar para encriptar y firmar digitalmente los datos. Que los protagonistas son propietarios de una Infraestructura de Clave Pública (PKI) que garantiza la asociación de la entidad de cada uno con su correspondiente clave pública por medio de un certificado. Asumimos también , que un protagonista cualquiera puede obtener en cualquier momento el certificado de cualquiera de los otros protagonistas y verificar su integridad y la validez de la clave pública asociada.

- Asumimos que todas las implementaciones son correctas, están certificadas como tales y que respetan la semántica del lenguaje. Esto debe ser certificado por una tercera parte independiente.

- Asumimos que todos los participantes de la infraestructura responden a una tercera parte de confianza (TTP - Trust Third Party), que se verá involucrada en caso de mal comportamiento de alguna de las partes.

El protocolo de intercambio de mensajes sería el siguiente:

Suponemos que el agente se inicia en el host A y en algún punto de la ejecución se le pide que migre a B. Como consecuencia se debe enviar a B tanto el código del agente (p) como el estado en el punto en el que migra (S_A). En ese momento A envía a B el siguiente mensaje, m_1:

A $\rightarrow$ B: A_S(A, B, K_A(p,S_A), A_S(A, i_A, t_A, H(p), TTP))

donde:

- A es el host de origen

- B es el host de destino

- K_A(p, S_A) son el código (p) y el estado (S_A) encriptados con una clave aleatoria elegida por el host de origen (A). Esta clave es conocida solamente por A hasta que B confirma que va a continuar con la ejecución del agente.

- A_S(A, i_A, t_A, H(p), TTP)) es el llamado token del agente. Este token contiene datos estáticos del agente y será usado en los sucesivos saltos del agente en su itinerario. Contiene la identidad de A, el identificador del agente (i_A), un timestamp (t_A) que indica el instante en el que se envió el agente, el valor de hash del código del agente (H(p)) y la identidad de la tercera parte de confianza que entrará en juego para resolver disputas en caso de haberlas (TTP). Este token va firmado por A.

Cuando B recibe este mensaje, usa la clave pública de A para comprobar la firma tanto en el código como en el token del agente. Como consecuencia, B puede aceptar o rechazar el mensaje. En cualquier caso B envía a A el siguiente mensaje, m_2:

B $\rightarrow$ A: B_S(B, A, i_A, H(m_1), M)

donde:

- M es la respuesta de B. Puede ser un rechazo o de compromiso de ejecución.

Cuando A recibe el mensaje, lo valida y comprueba el valor de M. Si M es un rechazo, la comunicación finaliza aquí. Si M es un compromiso de ejecución del agente, significa que es una petición implícita de la clave $K_A$con la que iban cifrados el código y el estado en el primer mensaje. En este caso A envía a B el siguiente mensaje, m_3:

A $\rightarrow$ B: A_S(A, B, i_A, B_p(K_A))

El mensaje va cifrado con la clave privada de A y la clave que B necesita para decodificar el código del agente va cifrada con la clave pública de B. B comprueba el mensaje con la clave pública de A (comprueba que es el firmante) y con su clave privada obtiene la clave K. Al estar firmada con la clave pública de B, solo B puede obtenerla. Una vez obtenidos todos los datos, B envía un mensaje de acuse de recibo a A, m_4, indicando que ha recibido la clave:

B $\rightarrow$ A: $B_S(B, A, i_A, H(m_3))$

Después de enviar el acuse de recibo a A, B decodifica el código con la clave que ha obtenido en el mensaje e inicia la ejecución del agente.

La ejecución del agente continua hasta que en algún punto del código se requiere que el agente migre al siguiente host del itinerario, C. En ese momento, B para la ejecución y envía a C dos mensajes consecutivos. Son los siguientes:

m_5:

B $\rightarrow$ C: $B_S(B, C, agent_A, H(T_B^p), H(S_B), t_B)$

donde:

- B es el host origen
- C es el host destino
- $agent_A$es el token del agente ($A_S(A, i_A, t_A, H(p), TTP)$))
- $H(T_B^p)$ es el valor de hash de las trazas producidas por el agente en su ejecución en el host origen, B.
- $H(S_B)$ es el valor de hash del estado del agente en el momento de la migración en el host origen B.
- $t_B$es el timestamp en B.

$m_5^{'}$:

B $\rightarrow$ C: $B_S(K_B(p, S_B), H(m_5))$

donde:

- B es el host origen
- C es el host destino
- K_B(p,S_B) son el código y el estado en B cifrados con una clave K_B escogida al azar por el host origen B.
- $H(m_5)$ es el valor de hash del mensaje m_5, enviado junto con este mensaje $m_5^{'}$.

El primer mensaje contiene los nombres del remitente y del receptor, el token del agente, el hash de las trazas generadas en el agente en B y el timestamp del envío de B a C. El segundo mensaje contiene el código y estado del agente cifrados y el valor de hash del primer mensaje.

Cuando C recibe los mensajes usa la clave pública de A para decodificar el token del agente y comprobar la identidad de A y el timestamp de envío del

agente. C comprueba las firmas de los mensajes y envía un mensaje de respuesta a B donde acepta o rechaza la ejecución de agente que se le envía. El mensaje de respuesta es el siguiente, m_6:

C $\rightarrow$ B: C_S(C, B, i_A, H(m_5, $m_5^{'}$), M)

donde:

- M es el mensaje de aceptación o rechazo a la ejecución.

Si M es un rechazo, se reinicia el agente y se devuelve un error como resultado de la sentencia que requería la migración. Si M es un mensaje de aceptación, significa que implícitamente requiere la clave para decodificar el código y estado del agente para poder continuar la ejecución. En ese caso B responde a C con el siguiente mensaje, m_7:

B $\rightarrow$ C: B_S(B, C, i_A, C_p(K_B))

donde:

- C_p(K_B) es la clave necesaria para decodificar tanto el código como el estado enviados en el mensaje $m_5^{'}$, encriptada con la clave pública de C, de forma que solo C podrá decodificar los datos enviados por B.

Cuando C recibe el mensaje, decodifica el código y estado. Comprueba que el código que ha obtenido no ha sido modificado calculando su valor de hash y comparándolo con el que llega en el token del agente. También calcula el hash del estado y lo compara con el valor que llega en el mensaje m_5. Una vez ha realizado las comprobaciones, si estas son correctas, C envía un mensaje de reconocimiento a B, m_8:

C $\rightarrow$ B: C_S(C, B, i_A, H(m_7))

Después C continúa con la ejecución del agente.

Este protocolo se repite para cada salto del agente hacia otro host en su itinerario hasta que el agente termina su ejecución. Al final de la ejecución, el último host (Z) contacta con el host origen (A) para enviarle el resultado de la ejecución. Z envía el siguiente mensaje a A, m_n:

Z$\rightarrow$ A: Z_S(Z, A, $agent_A$, H(T_Z^p), K_Z(S_Z), t_Z)

A comprueba la validez del mensaje y envía un mensaje a Z pidiendo la clave $K_Z$en un mensaje firmado, m_{n+1}:

A $\rightarrow$ Z: A_S(A, Z, i_A, H(m_n))

Z envía la clave K_Z a A usando la clave pública de A para cifrarla, de forma que solo A podrá decodificar el mensaje, m_{n+2}:

Z $\rightarrow$ A: Z_S(Z, A, i_A, A_p(K_z),H(m_{n+1}))

Después de examinar los resultados o los cargos por los recursos usados por el agente durante su ejecución, si A piensa que alguno de los hosts ha mentido, empieza el procedimiento de validación:

A, el host origen, pide a B, el primer host en el itinerario, que le envíe sus trazas, T_B^p. B no puede negar su participación debido a que envió el mensaje m_2 y debe enviar la traza a A. A comienza la simulación de la ejecución del agente con las trazas que le ha enviado B. En algún punto de la simulación, una sentencia pide la migración del agente al siguiente host del itinerario (C). En

este punto de la ejecución, el estado del agente es $S_B^{'}$. A pide a C sus trazas y una copia del mensaje m_5, que estaba firmado por B. C no puede negar la petición ya que B tiene el mensaje m_7 que indica la aceptación de C de ejecución del agente. A saca del mensaje m_5, los valores de hash de S_B y T_B^p. Con estos datos A puede comparar el hash enviado en los mensajes, $H(S_B)$, y el obtenido con la simulación, $H(S_B^{'})$. Además, A puede comprobar la integridad de las trazas enviadas por B anteriormente. Si las comprobaciones son correctas, B no ha mentido. Como consecuencia, A continua la simulación de la ejecución continuando con las trazas enviadas por el host C.

Este proceso continua hasta que el agente finaliza su ejecución. Al final, si las comprobaciones del estado enviado en los mensajes y el obtenido con la simulación ($S_Z^{'}$ y S_Z) y las trazas enviadas por Z concuerdan con el valor de hash contenido en el mensaje m_n, se puede constatar que el agente se ha ejecutado de forma correcta. Si, por el contrario, se encontrase alguna discrepancia durante la verificación de las trazas enviadas por el host X, entonces X habría mentido.

2.3. Ventajas del uso

Las ventajas que se obtienen al usar el mecanismo son las siguientes:

- Asegura que el código se ha ejecutado conforme a su especificación. Si se modificase el código las trazas recogidas en el fichero no concordarían.
- El propietario del agente está protegido frente a sobrecargas por el servicio realizado por el host en la ejecución. Al estar el código certificado por una tercera entidad se puede demostrar cuál es el código original a ejecutar por el agente.
- Si el agente se comporta de forma incorrecta, debido a la modificación del código por parte del host, es fácil saber quién es el responsable de la modificación, de los hosts del itinerario.

2.4. Desventajas del uso

Las desventajas que se obtienen al usar el mecanismo son las siguientes:

- Detección a posteriori: el mecanismo permite la detección de comportamientos maliciosos o ilegales a posteriori, es decir, después de la ejecución del agente. Esto implica que si el agente tiene un mal funcionamiento o ataca a otros recursos por haber sido manipulado su código no se podría evitar ni detectar con este mecanismo. Si es necesario detectar el ataque en el momento de la manipulación debería usarse otro mecanismo.
- No proporciona métodos para obtener culpables: el método no proporciona métodos para saber si realmente ha existido manipulación del código del agente. Se necesita un análisis posterior a la ejecución del agente de los

datos enviados como resultado para tener una sospecha de que realmente ha existido la manipulación. Es el propietario del agente el que sospecha y demanda las trazas al host sospechoso para probar si el host es realmente culpable o inocente.

- Necesaria infraestructura: los hosts involucrados en la ejecución (el itinerario que sigue el agente) deben tener la infraestructura necesaria para la gestión y distribución de claves.
- Necesaria la existencia de castigos: si no se castiga a los hosts infractores no tiene mucho sentido hacer la comprobación de culpabilidad. Además los hosts involucrados deben acatar las sanciones impuestas por una tercera parte independiente que modera la comunicación.
- Necesidad de gran capacidad de procesamiento: debido a que se usa criptografía asimétrica, que es muy lenta, es necesario que todos los hosts involucrados tengan gran capacidad de procesamiento. En otro caso los retardos serían demasiado elevados.
- Tamaño de los datos: las trazas generadas por un agente pueden ser muy grandes si este es complicado o tiene muchos datos de entrada. Esto requiere que los hosts dispongan de capacidad de almacenamiento para todos los agentes que ejecutan. Además no está definido el tiempo que deberían almacenarse las trazas para que el propietario de un agente pueda recuperar las trazas generadas en la ejecución en caso de sospecha.

Todas estas características limitan el alcance de aplicación del mecanismo.

Capítulo 3

Conceptos Previos

Para entender tanto las soluciones propuestas como la descripción del desarrollo llevado a cabo se deben tener conocimientos básicos de programación orientada a objetos, ya que Java es un lenguaje de programación orientado a objetos.

En este apartado se describen los conceptos básicos a los que se hace referencia en el resto del documento. En la fase de búsqueda de soluciones y en el de desarrollo e implementación de las mismas se hacen constantes referencias, ya que el funcionamiento del lenguaje hace que se deban adoptar unas soluciones u otras cuando se encuentran problemas a resolver.

Los conceptos que se van a explicar son los siguientes:

- Objeto
- Clase
- Herencia
- Interfaz
- Package

En las explicaciones se hace referencia al lenguaje de programación Java, ya que este es el que se ha seleccionado para el desarrollo de la solución al problema. Sin embargo, las características de Java no tienen por qué ser compartidas por el resto de lenguajes que se basan en la orientación a objetos.

3.1. Objeto

Un objeto es un conjunto de software que tiene un estado y un comportamiento relacionados. Los objetos de software se usan para modelar objetos del mundo real. Mediante software se modelan el estado (los atributos de un objeto y los valores que estas tienen) y el comportamiento (las acciones que un objeto puede llevar a cabo).

Los beneficios de la codificación mediante objetos son las siguientes:

- Modularidad: permite que el código de un objeto sea escrito y mantenido independientemente del código fuente de otros objetos.
- Ocultación de la información: el hecho de interaccionar solo con los métodos de los objetos hace que el detalle de la implementación interna permanezca oculta e inalterable para el resto del código.
- Reuso del código: un objeto que ya exista, incluso escrito por otro desarrollador de software, puede ser usado en un programa nuevo.
- Pluggability y facilidad de debugger: si un objeto en particular presenta problemas puede ser desenchufado (unplugged) y enchufar (plug) otro que lo reemplace. En lugar de reemplazar el programa completo, se reemplaza solamente la pieza del código que no funciona correctamente.

A continuación se detallan las características del estado y el comportamiento de un objeto.

3.1.1. Estado

El estado de un objeto lo definen los valores de los atributos definidos para una clase determinada. Estos atributos son comunes para todos los objetos del mismo tipo (es decir de la clase que define su tipo).

Los atributos pueden tener modificadores que hacen que se comporten de una forma o de otra. Los modificadores se indican en la definición de los atributos. Algunos de los modificadores más comunes son los siguientes:

- public: indica que el atributo es público y puede ser accedido (leído o escrito) directamente indicando su nombre (en el caso de java indicando: nombreClase.nombreAtributo).
- private: indica que el atributo es privado y no puede ser accedido directamente como en el caso anterior. Para cambiar o para obtener el valor de un atributo privado se deberán crear los métodos públicos que lo hagan (se deberá llamar a estos métodos para realizar los cambios).
- protected: indica que solo puede accederse como si fuese una atributo público por las clases que pertenecen al mismo paquete que la clase que lo define. En caso contrario es como si fuese un atributo privado, no puede ser accedido si no es por medio de métodos públicos que realicen las acciones pertinentes.
- final: indica que el atributo no puede cambiar de valor una vez ha sido inicializado. Es decir que el primer valor que se le da es el que va a tener durante toda su vida.
- static: indica que el valor que tiene no es independiente para cada objeto, sino que es común para la clase (que define el tipo del objeto).

Hay más modificadores que pueden ser aplicados en la definición de un atributo, pero no son tan comunes en los desarrollos, por eso no se han tenido en cuenta.

3.1.2. Comportamiento

El estado de un objeto lo definen los métodos definidos para una clase determinada. Estos métodos son comunes para todos los objetos del mismo tipo (es decir de la clase que define su tipo).

Los métodos pueden tener modificadores que hacen que se comporten de una forma o de otra. Los modificadores se indican en la definición de los atributos. Algunos de los modificadores más comunes son los siguientes:

- public: indica que el método es público y puede ser llamado directamente indicando su nombre (en el caso de java indicando: nombreClase.nombreMétodo(listaParámetros)).
- private: indica que el método es privado y no puede ser llamado directamente como en el caso anterior. Solo puede ser llamado desde otros métodos de la clase que lo define.
- protected: indica que solo puede accederse como si fuese una método público por las clases que pertenecen al mismo paquete que la clase que lo define. En caso contrario es como si fuese un método privado.
- final: indica que el método no puede redefinirse en clases que hereden de la clase que lo define. Es decir que el comportamiento no puede variar en las subclases que se van creando a partir de la que lo define.
- static: indica que no hay que crear una instancia de una clase para llamar al método. Se puede llamar directamente con: nombreClase.nombreMétodo(listaParámetros). Dentro del método solo se puede hacer referencia a atributos y otros métodos estáticos de la clase que lo defina.

Hay más modificadores que pueden ser aplicados en la definición de un método, pero no son tan comunes en los desarrollos, por eso no se han tenido en cuenta.

3.2. Clase

Una clase es un molde o prototipo a partir del cual se crea un objeto. En el prototipo se definen que características (atributos) y comportamientos (métodos) tendrá un objeto de esa clase.

En Java existen diferentes tipos de clases. El tipo de clase se indica con un modificador en la definición de la misma. Algunos de los modificadores más comunes son los siguientes:

- Abstract: indica que la clase es abstracta, lo que significa que no tiene todos sus métodos implementados. Uno o más métodos de la clase solamente están definidos. No se puede instanciar un objeto de este tipo de clase, para ello debería tener implementados todos los métodos. Estas

clases sirven para definir clases que tienen una parte del comportamiento común y una parte propia. La parte común se define en la clase abstracta (y solo deberá ser escrita una vez) y la parte propia se define en cada una de las clases que extiende de la clase abstracta.

- Privada: indica que no se puede acceder a la clase si no es desde el mismo fichero en el que está definida. En general, en Java, cada clase se define en un fichero separado y con el mismo nombre que la clase. Sin embargo, las clases privadas se pueden definir en un fichero en el que ya esté definida una clase pública. Será esta clase la única que pueda crear instancias de la clase definida como privada.
- Protected: indica que solamente ven (y por tanto solo pueden crear instancias) las clases que pertenecen al mismo paquete que en el que se define la clase protected. Al contrario que las clases privadas, las protected han de definirse en su fichero independiente como el resto de clases.
- Public: indica que cualquiera puede ver y crear instancias de este tipo de clases.

Hay más modificadores que pueden ser aplicados en la definición de una clase, pero no son tan comunes en los desarrollos, por eso no se han tenido en cuenta.

3.3. Herencia

La herencia es un mecanismo para organizar y estructurar el código. Cualquier clase puede heredar el comportamiento de otra clase. Una vez heredados los comportamientos se pueden modificar, quedando modificados solo en la clase que lo hereda. En este caso, el comportamiento de la clase original no se pierde y hay formas de llamar a este código en lugar que al implementado en la clase que hereda.

Permite no reescribir los comportamientos comunes de un conjunto de clases.

3.4. Interfaz

Una interfaz es un contrato entre una clase y el mundo exterior. La interfaz nos dice que comportamiento puede tener una clase que la implementa. La clase que implementa la interfaz está obligada a dar implementación a todos los métodos que esta define (si no debe declararse como abstracta).

En Java, una interfaz es una clase en la que define un conjunto de métodos, solamente el nombre y los parámetros que los métodos aceptan. Los métodos que se definen no tienen ninguna implementación, si algún método tuviese implementación debería declararse como clase abstracta, no como interfaz.

3.5. Package

Un package (paquete) sirve para organizar las clases e interfaces de una manera lógica de forma que sean más fáciles de manejar.

Las clases e interfaces se agrupan en diferentes contenedores con nombres lógicos que facilitan su comprensión. Normalmente se agrupan de forma jerárquica.

En los desarrollos de aplicaciones Java se han definir los paquetes a los que pertenecen las clases que se usan y que no están en el mismo paquete que la clase donde se usan, si no se tienen errores de compilación y el programa no se puede ejecutar.

Capítulo 4

Soluciones Propuestas

Como se ha descrito anteriormente, uno de los objetivos de este proyecto, era encontrar una forma de implementar lo enunciado en el documento [Vig98], donde se intenta obtener trazas de una aplicación cualquiera para luego poder reejecutarla con, únicamente, los binarios de la aplicación y el fichero de trazas generado en una ejecución. De las aplicaciones no tenemos los fuentes, donde se programó la funcionalidad de cada una de ellas, por tanto no es suficiente con modificar los fuentes y volver a compilar y empaquetar para tener el problema solucionado. Se ha investigado, por un lado, si ya se había realizado algún desarrollo similar aplicable a nuestro problema y, por otro, posibles formas de solucionarlo por si no se encontraba nada similar. Una vez comprobado que no había, disponibles, implementaciones para solucionar nuestro problema se pasó a la búsqueda de una forma de solucionarlo.

Para obtener una solución al problema presentado se han evaluado distintas posibles soluciones, todas ellas basadas en el desarrollo de componentes software que ofrecen el servicio que se busca, que es el de obtener trazas con los valores de entrada a las aplicaciones ejecutadas.

Las posibles soluciones tenidas en cuenta han sido las siguientes:

- Generación de clases propias para usar en la implementación de aplicaciones
- Uso de JVMTI (Java Virtual Machine Tool Interface)
- Uso de la API Reflection de Java
- Instrumentación de código

El objetivo del estudio es obtener la tecnología más indicada con nuestros requisitos. Cada una de las soluciones presenta una serie de ventajas y de inconvenientes. En algunos casos los inconvenientes encontrados indican que la tecnología no puede ser usada como solución.

A continuación se describe cada una de las soluciones, así como sus pros y contras y las conclusiones obtenidas del estudio realizado.

4.1. Generación de clases propias

La solución propuesta se basa en desarrollar clases propias que añadan la funcionalidad que se busca de obtener trazas de los datos de entrada a la aplicación.

4.1.1. Descripción

La JDK (Java Development Kit) de Java está formada por un conjunto de librerías que contienen clases útiles para el desarrollo de aplicaciones en Java. Estas clases definen los tipos básicos que pueden ser usados en el desarrollo y clases que proporcionan diversos servicios o que definen como han de ser estos para ser compatibles entre los diferentes desarrolladores o proveedores. Estas clases, además, forman parte de la JRE (Java Runtime Environment) y son las que se usan en la ejecución de aplicaciones.

Las clases que conforman la JDK no pueden ser modificadas directamente ya que si se modifican se contraviene a la licencia con la que se distribuye el código. Por esta razón, la idea sería generar clases nuevas con la misma funcionalidad que las clases originales, pero añadiendo la funcionalidad de obtención de trazas allí donde convenga.

Para obtener las clases nuevas, lo más fácil y flexible sería que las clases propias extendiesen (por medio de la herencia) de las clases originales. Con esto se conseguiría mantener la compatibilidad en el caso en que las clases originales cambiasen, cosa que puede ocurrir entre versiones distintas de la JDK. Sin embargo esto no es posible. Las clases de la JDK están definidas con el modificador "final" en su definición. Las clases finales en Java no pueden usarse para extender una clase , ya que esto provocaría un error de compilación en la clase que queremos generar.

Así, la solución se basaría en implementar clases propias, con la misma funcionalidad que las originales más la de obtención de trazas, pero desde cero, sin dependencia alguna de las originales. Para obtener realmente las trazas deseadas se debería asegurar que las aplicaciones se codifican con estas clases propias y no con las clases de la JDK de Java, ya que en caso contrario no se obtendrían los resultados esperados. El problema es que es difícil controlar, a priori, si en la implementación de una aplicación o programa se están usando realmente las clases nuevas o las que forman parte de la JDK. Además, para obtener las "copias" de las clases originales deberíamos saber que acciones realizan y codificarlas de forma parecida para no empeorar el rendimientos de las aplicaciones que las usen (se supone que las clases de la JDK están codificadas de la forma más óptima posible en cuanto al rendimiento se refiere).

Esta fue la primera idea que surgió para solucionar el problema propuesto y la más sencilla de implementar. Sin embargo es poco flexible. Uno de los principales problemas radica en que la JDK de Java va modificándose con las nuevas versiones que van saliendo, por tanto se deberían ir actualizando las clases para mantener la compatibilidad con el código original.

Además, hay otras desventajas, ya que con esta solución no se soluciona

el caso de obtener trazas de los datos de entrada de la aplicación. Habría que generar una clase solo para la obtención de las trazas de los datos de entrada a la aplicación, ya que no es una clase genérica la que lee los datos de entrada, sino que es la clase que el desarrollador defina como punto de entrada. En el caso de acceso a datos de ficheros no habría problema, ya que son clases no abstractas ni interfaces, y son las mismas para todos los tipos de ficheros y plataformas (Windows, Unix, ...). Sin embargo en el caso de bases de datos deberíamos crear clases propias independientes para cada uno de los fabricantes de gestores de bases de datos, ya que son los propietarios los que desarrollan los drivers necesarios para el acceso a los datos de sus bases de datos y estos son desarrollados por cada fabricante de forma independiente (se basan en la interfaz que define JDBC para los métodos que deben ser implementados, sin embargo, el como se hacen o si hay más métodos que los definidos en la interfaz es libre para cada uno de los fabricantes). Al igual que con las clases de la JDK, se debería revisar en cada una de las versiones del driver que no se ha perdido compatibilidad.

Esta solución es factible técnicamente. Por lo menos, la generación de las clases propias es factible. Por esta razón queda como solución a adoptar en el caso en que el resto de posibles soluciones no sean factibles por alguna razón, ya sea técnica o de rendimiento.

4.1.2. Mejoras: Combinación con instrumentación de código

Esta solución se puede combinar, para obtener una solución mejorada, con el uso de instrumentación de código, ya que unas de las características de la instrumentación de código es la de cambiar, dentro de la definición de una clase, las llamadas a una clase específica por cualquier otra clase. En nuestro caso, en lugar de obligar al programador a usar las clases nuevas, se trataría de coger su código desarrollado y mediante la instrumentación hacer el cambio de las clases de la JDK por nuestras clases propias que incluyen la obtención de trazas.

Esta mejora a la solución propuesta inicialmente, sigue siendo poco flexible por el hecho de tener que comprobar que se mantiene la compatibilidad entre las clases generadas y los cambios que se produzcan entre versiones de la JDK. Sin embargo evita el obligar al desarrollador a tener que programar con clases que no son las naturales del lenguaje, y, además, asegura que realmente se estarán usando las nuevas clases en la totalidad del código y que se obtendrán las trazas correctamente.

4.2. JVMTI

JVMTI (JVM Tool Interface) es una interfaz de programación usada por herramientas de desarrollo y de monitorización.

4.2.1. Descripción

Proporciona una forma de inspeccionar el estado y de controlar la ejecución de las aplicaciones que se ejecutan en la JavaTM Virtual Machine (JVM). El tipo de herramientas que usan los servicios que proporciona esta interfaz están entre los siguientes: herramientas de debbugger, monitorización, análisis de threads, etc...

Esta interfaz puede no estar disponible en todas las implementaciones de la JVM.

JVM TI funciona en dos modos:

- Modo agente: un cliente de JVM TI, a partir de ahora llamado agente, puede ser notificado de ocurrencias interesantes por medio de eventos.
- Modo consulta: JVM TI consulta y controla las aplicaciones por medio de un conjunto de funciones, ya sea en respuesta a eventos o independientemente de ellos.

Los agentes se ejecutan en el mismo proceso que la máquina virtual que ejecuta la aplicación que está siendo examinada y se comunica directamente con ella.

JVM TI está contenida dentro de JPDA (Java Platform Debugging Arquitecture). Es una herramienta a muy bajo nivel, donde los agentes se programan directamente en lenguaje nativo que soporte definiciones de C o C++. JPDA contiene otras interfaces de más alto nivel que son más adecuadas en muchos casos que JVM TI.

Las definiciones de funciones, tipos de datos y definiciones de constantes están definidas en el fichero de includes jvmti.h. Para poder usar las definiciones se ha de añadir el directorio include de J2SE en el path de includes y añadir #include <jvmti.h> al código fuente.

Es posible pasar argumentos al agente en la línea de comandos. Estos argumentos de la línea de comandos se usan en el arranque de la VM para cargar y ejecutar de forma correcta los agentes. En ellos se indican la librería que contiene el agente y las opciones (en forma de Strings) que se han de usar en la configuración y el arranque de los mismos.

4.2.2. Instrumentación de código

La interfaz proporciona soporte para instrumentación de bytecode, que es la habilidad de alterar las instrucciones de bytecode de la Java virtual machine. Estas alteraciones se usan, típicamente, para añadir eventos al código de un método. Debido a que los cambios son únicamente para añadir código, el comportamiento de la aplicación o su estado no se ven modificados. El código del agente está en bytecodes estándar, por tanto la VM puede ejecutarse a la velocidad más alta, optimizando no solo el programa que se quiere ejecutar sino también la instrumentación realizada sobre el código original. La instrumentación puede ejecutarse completamente en código Java o puede llamar al agente nativo y puede servir para mantener contadores o para muestrear eventos de forma estadística.

Hay tres formas de insertar la instrumentación:

- Instrumentación estática: los ficheros de clase son instrumentados antes de ser cargados en la VM. Este proceso suele ser delicado y el agente no puede saber el origen de las clases que han de ser cargadas.
- Instrumentación en tiempo de carga: cuando una clase es cargada por la VM, los bytes del fichero de clase se envían al agentes para que sean instrumentados. Este mecanismo proporciona un acceso eficiente y completo a la instrumentación a un tiempo.
- Instrumentación dinámica: una clase que ha sido cargada y posiblemente está en ejecución es modificada. Las clases pueden ser modificadas múltiples veces y pueden volver a su estado original. Este mecanismo permite instrumentaciones con cambios durante el curso de la ejecución del programa.

4.2.3. Conclusiones

El uso de JVM TI hace que las aplicaciones no sean portables, ya que se basa en agentes codificados en lenguajes nativos y no interpretados, como pueden ser C o C++. Aunque esto hace que el rendimiento de las aplicaciones sea óptimo, incluso añadiendo instrumentación a las clases, pero la pérdida de portabilidad es suficiente para desecharlo como solución al problema. En el caso de agentes móviles, la portabilidad es un requerimiento, ya que si no, no podría migrar de host en host para su ejecución.

Además la instrumentación que se permite es la de añadir código extra a los métodos y no se indica si hay la posibilidad de obtener valores de variables en tiempo de ejecución.

4.3. Uso de la API Reflection de Java

La API Reflection de Java no es una herramienta de instrumentación de código propiamente dicha. Sin embargo, tiene la habilidad de examinar o modificar el comportamiento en runtime de las aplicaciones que se ejecutan en la Java Virtual Machine (JVM). Es por esta habilidad que se ha escogido para intentar obtener las trazas de una aplicación cualquiera (incluso sin tener el código de la misma).

Esta API tiene tanto ventajas como desventajas.

Las ventajas son las siguientes:

- **Extensibilidad:** Hace posible la creación de instancias y su uso en clases definidas por el usuario. Gracias a esta ventaja podemos obtener el método de entrada y llamarlo después de hacer los pasos que se crean necesarios (en este caso recoger los valores de los parámetros de entrada).

- **Inspección de código:** Permite hacer inspección de código y recoger tanto los métodos como las variables definidas en una clase. También posibilita la recogida de valores de las variables en tiempo de ejecución.

Las desventajas son las siguientes:

- **Performance overhead:** Debido a que el uso de Reflection implica el uso de tipos que se resuelven de forma dinámica, ciertas optimizaciones de la JVM no se pueden hacer. De esta forma las operaciones se hacen más lentas y, por tanto, no debe usarse en secciones que se llaman de forma recurrente. En nuestro caso, en la recogida de parámetros inicial solo se hace una vez al arrancar la aplicación, con lo que la pérdida de performance será la mínima posible.
- **Restricciones de seguridad:** Para usar Reflection se requieren permisos en tiempo de ejecución que pueden no estar presentes en el momento de la ejecución de la aplicación bajo un mánager de seguridad (Security manager). Las aplicaciones standalone, como las que se usan en este proyecto, por defecto no tienen manager de seguridad.
- **Exposición de los datos internos:** Permite acciones que podrían ser ilegales en un entorno no reflectivo, como acceder a métodos privados, que podrían tener efectos colaterales. El código reflectivo rompe la abstracción de las clases y por tanto puede cambiar el comportamiento con las futuras versiones de la plataforma.

Otra de las desventajas que se ha encontrado es que se hace necesario llamar a la aplicación de forma indirecta, a través de una clase genérica que es la que hace el trabajo llamar a la clase que inicia la aplicación que se quiere ejecutar y de almacenar las trazas en el fichero.

4.4. Instrumentación de Código

Las clases Java (el código con las instrucciones de lo que debe hacer una aplicación o programa desarrollado en Java) son compilados en ficheros de clase binarios y portables. Estos ficheros contienen código interpretado que está compuesto por un conjunto limitado de instrucciones (byte code). Los ficheros son cargados por el cargador de clases de la JVM y ejecutados para obtener los resultados deseados.

Por instrumentación de código se entiende la habilidad de modificar estos ficheros binarios de clase antes de que sean cargados por la JVM para su ejecución y así modificar su comportamiento, para añadir instrucciones de forma dinámica al código original.

La instrumentación se puede realizar de dos formas, en función a si almacenan o no los resultados de la instrumentación de una aplicación:

- Estáticamente: al instrumentar una clase estáticamente, se modifica su código, añadiendo las instrucciones deseadas y se almacena el resultado en un fichero de clase que reemplazará en la ejecución al fichero original.

- Dinámicamente: al instrumentar una clase de forma dinámica, se modifica su código justo antes de la carga en la JVM, de forma que el código resultante de la instrumentación no se almacena y es válido únicamente durante el tiempo de ejecución de la aplicación que se quiere instrumentar.

También, en función del lenguaje usado en la instrumentación, tenemos los siguientes tipos de instrumentación:

- A bajo nivel: en este caso las instrucciones que se quieren añadir con la instrumentación se han de indicar como byte codes (en binario). Este tipo de instrumentación permite usar cualquier instrucción del lenguaje Java. Sin embargo, para realizar una instrumentación de este tipo se deben poseer conocimientos profundos sobre la JVM y la estructura de un fichero class, para no cometer errores.

- A alto nivel: en este caso, las instrucciones a añadir con la instrumentación se indican directamente en Java. Este tipo es menos flexible y tiene sus limitaciones. No todas las herramientas soportan este tipo y muchas dependen de un compilador propio que define las limitaciones que tiene el código que se puede indicar. En este caso, la ventaja principal, es que no se requieren conocimientos profundos sobre la JVM ni sobre la estructura de un fichero clase de java. Con tener conocimientos del lenguaje es suficiente para obtener una instrumentación satisfactoria.

Existen multitud de herramientas en Internet para instrumentar clases. Algunas de estas herramientas son las siguientes:

- Javassist
- BECL
- java.lang instrument

No se ha estudiado el detalle todas las herramientas indicadas. Asimismo, muchas de ellas han dejado de ser soportadas por sus creadores.

4.4.1. Java: paquete java.lang.instrument

En la versión 1.5 de la JDK de Java apareció el paquete java.lang.instrument. Este paquete proporciona servicios que permite que agentes programados en Java instrumenten programas que se ejecutan en la JVM (Java Virtual Machine). Cuando se arranca la JVM se le indica, mediante un modificador en la línea de comandos, la clase del agente a arrancar y las opciones con las que este se arranca.

4.4.1.1. Descripción

El uso de esta librería se basa en el desarrollo de agentes que implementan una interfaz predefinida que permite la instrumentación de clases Java, antes

de ser definidas en al JVM. Para ello, los agentes deben implementar un método estático y público (en principio muy similar al main usado como punto de entrada de las aplicaciones Java) con la siguiente firma:

public static void premain(String agentArgs, Instrumentation inst);

Después de inicializar la JVM, se llama a cada uno de los métodos premain en el orden en el que han sido especificados. Una vez se han llamado a todos los premains, se llama al método main, punto de entrada de la aplicación que se quiere ejecutar. Cada método premain debe acabar bien (es decir, sin que se produzca ningún error o capturando los errores que se produzcan) para que la secuencia de arranque pueda proseguir hasta arrancar la aplicación que realmente se ha de ejecutar. La clase del agente debe ser cargada por el mismo cargador de clases que cargan las clases que contienen el método main de las aplicaciones. El método premain correrá con la misma seguridad y las mismas reglas del cargador de clases que el método main de la aplicación.

No hay ninguna restricción en las acciones que se pueden realizar dentro del método premain de un agente. Cualquier acción que pueda llevarse a cabo dentro del main de una aplicación, incluida la generación y el arranque de threads, es legal dentro de la función premain del agente.

A cada agente se le pasan sus opciones vía el parámetro agentArgs. Las opciones se pasan al agente como un String, y es el el que debe realizar el parseo de los valores para obtener los tipos necesarios para su correcto funcionamiento.

Si el agente no se puede resolver (por ejemplo, porque la clase del agente no puede ser cargada, o porque la clase del agente no tiene el método premain conforme a la especificación), la JVM abortará su ejecución.

En las JVM con una interfaz de línea de comandos, los agentes se especifican añadiendo el siguiente modificador a la línea de comandos que la arranca:

-javaagent:jarpath[=options]

donde:

- jarpath: es el path al JAR que contiene la clase del agente.
- options: son las opciones del agente.

Si se quieren usar múltiples agentes, el modificador debe ser usado múltiples veces en la misma línea de comandos, cada una con uno de los agentes a arrancar. El JAR que contiene al agente debe conformar la especificación de los ficheros JAR. Los siguientes atributos de manifest son definidos para un fichero JAR de un agente:

- Premain-Class: es la clase que contiene el método premain. Este atributo es requerido y, si no está presente, la JVM abortará su ejecución.
- Boot-Class-Path: es una lista de paths donde buscar el cargador de clases. Este atributo es opcional.
- Can-Redefine-Clases: es un booleano (true o false) que indica si el agente necesita la habilidad de redefinir clases. Cualquier otro valor que no sea "true" será considerado como "false". El atributo es opcional y por defecto es "false".

El fichero JAR del agente es adjuntado al classpath.

4.4.1.2. Uso de la librería

El uso de esta librería se basa en el uso en dos interfaces y una clase:
Las interfaces son las siguientes:

- Instrumentation: esta clase provee servicios necesarios para instrumentar código desarrollado con lenguaje de programación Java. Por instrumentación se entiende la adición de byte-codes a los métodos con el propósito de recoger datos para ser utilizados por herramientas. Debido a que los cambios son puramente aditivos, estas herramientas no modifican el estado o comportamiento de la aplicación. Ejemplos de herramientas incluyen agentes monitorizadores, analizadores y loggers de eventos. La única forma de acceder a una instancia de la interfaz Instrumentación es arrancando la JVM de forma que se indique la clase del agente. La instancia de Instrumentation se pasa al método premain de la clase del agente. Una vez el agente tiene la instancia de Instrumentation, puede llamar a métodos en la instancia en cualquier momento.
- ClassFileTransformer: Un agente proporciona implementación a esta interfaz para poder instrumentar ficheros de clase. La transformación ocurre antes de que la clase sea definida en la JVM. Un fichero de clase no tienen porque estar en un fichero físico de forma obligatoria, con cumplir con la especificación de ser una secuencia de bytes con formato de fichero de clase es suficiente para ser considerada como tal. La interfaz tiene un método, que hay que definir en la clase del agente y es el que se utiliza para cambiar la definición de una clase.

La clase es la siguiente:

- ClassDefinition: esta clase sirve como parámetro al método redefineClasses de la interfaz Instrumentation. Sirve para ligar la clase que se va a redefinir con los bytes de la nueva clase.

4.4.1.3. Ventajas e Inconvenientes

La ventajas de usar esta librería son las siguientes:

- Está incluida en la JDK, por lo tanto no se ha de añadir ningún JAR externo para ejecutar la aplicación.

Las desventajas de usar esta librería son las siguientes:

- Está pensada solamente para añadir código en la creación de agentes de monitorización o loggeo de excepciones.
- Se ha de indicar al arrancar la JVM el modificador para que se arranque también el agente (con la clase donde este está definido y las opciones con las que se arranca).

- Es una librería a bajo nivel, es decir, que se han de indicar los bytes correspondientes al bytecode de una clase Java para poder hacer la instrumentación.

- No se modifican partes concretas de una clase, sino que debe instrumentarse la clase completa (se cambia un array de bytes que contiene el código por el array de bytes que se le pasa a la función correspondiente). Esto hace pensar que se necesita saber como es la clase antes de la modificación y en nuestro caso no disponemos del código original, sino que tenemos solamente las clases compiladas (ficheros .class), sobretodo en el caso de acceso a bases de datos, donde el código es proporcionado por el fabricante. En nuestro caso queremos modificar unas sentencias y métodos determinados independientemente de donde se encuentren realmente en el código e independientemente de como sea la clase o conjunto de clases donde estos se usan.

4.4.2. Javassist

Javassist (JAVA programming ASSISTant) es una librería potente en el campo de la ingeniería de byte code. Permite que los desarrolladores añadan métodos a una clase compilada, modifiquen el cuerpo de un método, y así muchas otras modificaciones sobre el código ya compilado (no es necesario tener el código fuente para realizar la modificación de las clases). La característica más importante de Javassist es que permite realizar las modificaciones sobre las clases sin necesidad de conocer con detalle el funcionamiento del byte code de Java o de la estructura de un fichero de clase (.class).

La ingeniería de byte code se usa para manipular y modificar clases Java compiladas y también para la creación de nuevas clases de forma programática. Las manipulaciones se pueden llevar a cabo de dos formas:

- En tiempo de ejecución: las clases son modificadas justo antes de ser cargadas por el cargador de clases y el byte code generado no se almacena en un fichero, si no que es usado para la ejecución actual y después se pierde. La desventaja es que, la aplicación a ejecutar no puede ser llamada directamente, si no que se llama de forma indirecta, por medio de otra clase que es la que realiza las manipulaciones necesarias sobre el código y luego realiza la llamada a la aplicación que realmente queremos ejecutar.

- En tiempo de compilación: las clases se modifican sin necesidad de ser ejecutadas, en tiempo de compilación, y el byte code resultado de la manipulación se almacena en ficheros para ser usados posteriormente en las ejecuciones del código. Esto permite no tener que ejecutar la clase mediante una clase intermedia (que en el caso anterior seria la que realiza la llamada a las clases que realizan las modificaciones). Sin embargo, si se modifican clases de sistema es necesario indicar a la JVM donde están las clases instrumentadas a utilizar para que realmente sean usadas. En nuestro caso no instrumentamos clases de este tipo.

4.4.2.1. Descripción

La API de Javassist tiene dos partes:

- Alto nivel: con los servicios disponibles en esta parte de la librería se pueden realizar modificaciones directamente usando bloques de código Java. El conjunto de instrucciones a usar es limitado, ya que depende de un compilador interno que proporciona la misma librería. Sin embargo evita tener que conocer en detalle aspectos técnicos que no todos los desarrolladores conocen en detalle, como puede ser la estructura de un fichero de clase (.class).
- Bajo nivel: en esta parte de la librería, los bloques de código a modificar se han de indicar en binario (byte codes). Es más potente que la parte de alto nivel, ya que permite cualquier instrucción de código que sea legal dentro del lenguaje. Sin embargo obliga a conocer el conjunto de instrucciones (byte codes) y la estructura que ha de tener un fichero de clase (.class) para poder realizar la instrumentación de forma satisfactoria.

La API es similar a la API Reflection de Java, ya que permite ver la estructura que tiene una clase antes de ser cargada por el cargador de clases. Incluye las siguientes clases que representan partes de una clase:

- CtClass: representa una clase compilada. Una vez obtenida una instancia de esta clase podemos obtener la lista de métodos y constructores con sus parámetros correspondientes. También podemos ver los atributos definidos y los modificadores de estos y de los métodos definidos (si son públicos, privados, abstractos, etc...). Podemos saber de que clase extiende y que interfaces implementa.
- CtMethod: representa un método definido en una clase. Una vez obtenida una instancia se pueden ver los parámetros del método, así como obtener la clase en la que está definido (CtClass). También se puede añadir eventos que saltan cada vez que se encuentra un método y que permite instrumentarlo de forma independiente en función de los filtros que apliquemos.
- CtField: representa un campo definido en una clase. Una vez obtenida una instancia de esta clase podemos ver los modificadores con los que ha sido definido (público, privado, etc...), así como la clase a la que pertenece.

La API Javassist no proporciona métodos para crear nuevas instancias, invocar métodos o acceder a valores de campos, ya que los objetos CtClass representan clases que aun no han sido cargadas. Sin embargo si que proporciona métodos para modificar las definiciones de los métodos. Al ejecutar una aplicación se usarán las clases modificadas existentes.

La API se puede usar en conjunto con un ClassLoader para modificar las clases en tiempo de ejecución. Se incluye un cargador de clases por defecto, aunque cada desarrollador puede implementar uno a su gusto y necesidad. Con esto se evita la necesidad de enviar las clases generadas previamente en el momento de la ejecución.

4.4.2.2. Uso de la librería

Para obtener instancias CtClass de clases compiladas existe la clase ClassPool. Esta clase es una factoría de objetos CtClass, es decir que se encarga de crearlos. Para ello, solamente necesita saber el nombre de la clase y el paquete al que pertenece. Con estos datos, el objeto ClassPool busca entre las clases que ya ha cargado si la tiene y si es así devuelve la que tiene almacenada. Si no la tiene, la carga y después la devuelve una vez la tiene en su lista. Así, el objeto ClassPool seria como un array de clases que se han ido cargado, y también modificando, y se mantienen en memoria para mantener los cambios que se hayan hecho sobre ellas.

Proporciona un conjunto de variables especiales para obtener valores en los bloques de código instrumentados. La lista de variables especiales, en el caso de un método, es la siguiente:

- $0,$1,$2,...: Representan el objeto "this" y el conjunto de parámetros que son pasados al método.
- $args : Array de objetos (Object []) con los valores de los parámetros que se pasan al método en la llamada.
- $$: Representa a todos los parámetros. Es decir que es equivalente llamar a una función como f($$) que como f($1,$2,...).
- $cflow : cflow variable. Esta variable vale para métodos que se llaman de forma recursiva, para saber cual es la profundidad de la ejecución actual (o cuantas veces ha sido llamado el método de forma recursiva).
- $r : tipo del resultado del método. Se usa en operaciones de Cast.
- $w : es el tipo del wraper de un parámetro. Se usa en operaciones de Cast.
- $_ :es el valor del resultado. Se puede obtener y setear para modificar el resultado de un método.
- $sig : array de java.lang.Class que contiene los tipos formales de los parámetros de entrada del método.
- $type : objeto de tipo java.lang.Class que indica el tipo formal del resultado del método.
- $class : objeto de tipo java.lang.Class que indica la clase que está siendo modificada.

Estas variables se usan dentro del bloque de código que se usa para modificar el cuerpo de un método para cambiar su comportamiento, los valores de los parámetros o el del resultado que devuelve. La desventaja de usar estas variables especiales es que obliga a añadir el paquete javassist.runtime para que la ejecución pueda llevarse a cabo.

4.4.2.3. Ventajas e inconvenientes

Las ventajas son las siguientes:

- En el caso de la parte de alto nivel, se pueden indicar los bloques de código directamente en lenguaje de programación Java.
- Permite realizar modificaciones sobre las clases sin conocer la estructura de un fichero de clase o el conjunto de instrucciones permitidos (byte code) y su semántica.
- La pérdida de rendimiento depende solo de los bloques de código que se añadan al código original.

Los inconvenientes de usar esta librería son los siguientes:

- En el caso de la parte de alto nivel, el rango de instrucciones Java a incluir en los bloques de código que se indican están limitados. Sin embargo las limitaciones son pocas.
- En el caso de usar variables especiales, necesita librerías de la API de Javassist en tiempo de ejecución.

4.4.3. BCEL

BCEL (Byte Code Enginnering Library) da la posibilidad a los usuarios a analizar, crear y manipular ficheros Java binarios (.class). Las clases contenidas en estos ficheros son representadas por objetos que contienen toda la información simbólica de la clase dada: métodos, campos y las instrucciones byte code.

Estos objetos pueden ser leídos de un fichero existente, transformados por un programa (cargándolo en tiempo de ejecución) y vueltos a escribir en ficheros otra vez. La librería también permite crear clases desde cero en tiempo de ejecución.

Una de las desventajas del uso de esta librería es que es necesario conocer la estructura interna de la JVM (Java Virtual Machine) y del formato de los ficheros .class, ya que tienen un formato predefinido. Otra de las desventajas es que no se han realizado muchos desarrollos para mejorar la librería en los últimos años, sin embargo se usa con éxito en compiladores, optimizadores, ofuscadores de código, generadores de código y herramientas de análisis.

Debido a que, como se ha comentado, requiere conocimientos de la estructura tanto de la JVM como de la estructura de los ficheros .class, no ha sido usada en los desarrollos del presente proyecto.

4.4.4. Conclusiones

Existen múltiples herramientas de instrumentación de código. Más incluso que las que se han enumerado en el presente capítulo. La mayoría de ellas solo permiten la modificación de código a bajo nivel (es decir, indicando directamente los byte codes necesarios). La única que permite indicar las modificaciones a

realizar como código Java es la API Javassist. Por esta razón es la herramienta seleccionada para el desarrollo de la solución.

4.5. Conclusiones, soluciones y herramientas seleccionadas

Todas las posibles soluciones al problema presentas ventajas e inconvenientes.

En el caso de JVM TI, el hecho de desarrollar los agentes en código nativo hace que las aplicaciones pierdan la portabilidad. Esto, en el caso de agentes móviles, es un requisito básico para que puedan ir migrando de host en host y ejecutando su código. Si el agente no puede migrar a un host porque no es compatible el código con la plataforma no podrá ejecutarse el código correctamente o no podrá seguir el itinerario que se le ha marcado. Además requiere indicar el agente (la clase a la que pertenece) que realiza la instrumentación en el arranque de la JVM, lo que no es posible que siempre vaya a poder realizarse, ya que es el host remoto el que tiene el control sobre el modo de ejecución del código. Por estas razones esta posible solución queda descartada.

La opción del uso de la API Reflection de Java se ha tenido en cuenta solamente en el problema de obtención de los datos de entrada de la aplicación. Esta API penaliza mucho el rendimiento de las aplicaciones en las que se usa y no está indicada para tareas que se repiten constantemente en la aplicación. En el caso de los datos de entrada a la aplicación solo se ejecuta una vez, con lo que la pérdida de rendimiento no es mucha. Sin embargo en el resto de soluciones no se sabe cuantas veces se va a realizar la misma tarea repetitiva y con esta tecnología se puede añadir mucho retardo en las ejecuciones de las clases resultantes.

De las dos restantes, la primera opción a tener en cuenta es la instrumentación de código, ya que es la más flexible y al no obligar a nada al desarrollador es la que va a tener menor fuente de errores. Sin embargo, la de instrumentar el código es la más complicada de implementar ya que hay que tener muy claro en que puntos se ha de añadir el código de obtención de trazas para que la solución sea satisfactoria. Además obliga, en el caso de usar características avanzadas de las librerías (como el uso de las variables especiales de Javassist), a añadir al classpath librerías necesarias para la ejecución de las aplicaciones resultantes de la instrumentación.

La primera opción es la menos flexible, pero es siempre factible técnicamente. Aunque esta solución implica obligaciones a los desarrolladores. Para que funcione se debe asegurar que se usan las nuevas clases generadas (cosa que puede evitarse si se usa la instrumentación de código para realizar el intercambio de clases). Uno de los inconvenientes importantes es que obliga a revisar continuamente las clases generadas ya que las originales pueden modificarse con cada nueva versión de la JDK que aparece.

En el desarrollo de este proyecto se ha optado por usar instrumentación de código y más concretamente la librería Javassist por los siguientes motivos:

- Es la solución más flexible
- JVM TI es descartada por falta de portabilidad, característica básica para el correcto funcionamiento de un sistema de agentes móviles.
- La API Reflection queda descartada como solución, excepto en el caso de los datos de entrada por la pérdida de rendimiento que comporta su uso.
- Si no es factible técnicamente se puede usar la instrumentación para, combinándola con la opción de la generación de clases propias, sustituir las clases originales de la JDK por las generadas nuevas que añaden la obtención de trazas, que sabemos que si es técnicamente factible.
- Javassist permite instrumentación de código a alto nivel, es decir indicando el código a añadir con la instrumentación directamente en Java. En el caso en que se compruebe que el resultado es satisfactorio se puede probar a instrumentar el código con la parte de bajo nivel, donde no hay restricciones en el código que se inserta.

Capítulo 5

Desarrollo e Implementación de la solución

5.1. Objetivos

Como se ha comentado anteriormente, el objetivo del proyecto es implementar una posible solución a los problemas de seguridad de los agentes móviles. La idea es obtener trazas de la ejecución de los agentes durante su ejecución en los hosts remotos, tanto si se confía en ellos como si no se confía. Si se detectan comportamientos erróneos o extraños en la ejecución o se sospecha del host, se le puede pedir el fichero de trazas generado y volver a ejecutar en local la aplicación con el código original y los datos almacenados en el fichero de trazas. Comprobando los resultados obtenidos en la ejecución del host y en ejecución de la máquina local se puede saber si la ejecución ha sido correcta o si el host externo ha intentado atacar al agente de algún modo para beneficiarse del resultado de la ejecución.

Un agente se compone de un segmento de código y de un estado. El segmento de código se supone estático en el tiempo (es decir, que el agente no puede modificar su código durante la ejecución). El estado es variable en el tiempo durante la ejecución. El segmento de código se compone de una secuencia de sentencias que pueden ser blancas o negras. Una sentencia blanca es aquella que modifica el estado del agente basándose solo en valores de variables internas del agente. Una sentencia negra modifica el estado del programa usando información recibida desde el exterior del entorno de ejecución. En nuestro caso nos interesará almacenar trazas de las sentencias negras, ya que son las que llegan desde el exterior y en el entorno local (reejecución) no las podríamos tener.

Como solución, se propone la generación de un fichero de trazas de las sentencias negras del segmento de código del agente. Una traza de ejecución se compone de una secuencia de pares, donde el primero de ellos representa un identificador único de la sentencia traceada y el segundo es la firma (la operación/es que realiza). En el caso de sentencias negras, la firma contendrá los

nuevos valores que se asignarán las variables internas como consecuencia de la sentencia.

Los modelos de sentencia negra que pueden darse son los siguientes:

- Entradas del usuario: ratón, teclado, pantalla táctil...
- Datos de entrada de la aplicación
- Acceso a datos almacenados en la misma máquina o máquinas remotas:
  - Datos almacenados en ficheros
  - Datos almacenados en bases de datos
- Streams de datos: los streams de datos representan un origen de datos. Los diferentes orígenes de datos pueden ser de los siguientes tipos:
  - Ficheros almacenados en disco
  - Datos provenientes de dispositivos
  - Datos provenientes de otros programas
  - Datos provenientes de arrays de memoria
  - ...

Los agentes no tienen interacción con un usuario ya que son aplicaciones autónomas con control sobre sus acciones. Por esta razón no tendremos en cuenta la primera opción de la lista (entradas de usuario) ya que no se van a dar. Respecto a los streams de datos, solo se han tenido en cuenta en el desarrollo y pruebas los datos provenientes de ficheros almacenados en disco, ya que el resto de posibles orígenes se tratan programáticamente de la misma manera.

Para llegar al objetivo se barajaron distintas opciones. El problema principal era el no tener el código fuente de las clases, por tanto no se podía modificar el código y volver a compilar los fuentes para obtener clases que dejasen trazas. Se necesitaba una solución para modificar los binarios (código compilado o bytecode Java).

Con las especificaciones que se tenían, se realizó un estudio para encontrar la solución más adecuada al problema. Finalmente se seleccionó usar la técnica de instrumentación de código y más específicamente la librería de instrumentación de código Javasssit. Esta API es la que se ha usado en la mayor parte del desarrollo del proyecto. Esta API tiene dos partes, la de alto y la de bajo nivel, pero para el desarrollo se ha usado solamente la parte de alto nivel.

5.1.1. Alcance del proyecto

Debido a que la obtención de trazas es dependiente del lenguaje de programación en el que se haya desarrollado la aplicación, en este proyecto se ha buscado un método para las aplicaciones desarrolladas con el lenguaje de programación Java.

Java es un lenguaje de programación orientado a objetos. Esto significa que se compone básicamente de objetos que se envían mensajes entre si para llevar a cabo las tareas que ha de realizar la aplicación. Este lenguaje de programación requiere que todas las variables se declaren antes de se usadas. En la declaración se indica el tipo de la variable y se inicializa con un valor si es necesario. Java tiene diferentes tipos de variables: tipos primitivos y objetos. En este proyecto solo se han tenido en cuenta las sentencias que tratan con tipos de datos primitivos de Java o de tipo String.

Hay 8 tipos de datos primitivos, son los siguientes:

- byte
- short
- int
- long
- float
- double
- boolean
- char

El tipo de dato String no es de tipo primitivo, sino que es de tipo Objeto, pero también se ha tenido en cuenta porque representa una cadena de caracteres (char) y es un tipo de dato muy común en el desarrollo de aplicaciones. Además tiene un tratamiento especial, ya que no es necesario declararlo como un objeto y crearlo como un constructor, sino que puede ser definido como una variable de tipo de dato primitivo.

Además suponemos en todos los casos que el flujo de la aplicación es el normal y que no se producen excepciones que lo hagan variar. El tratamiento de excepciones se tendría que indicar de alguna manera en el log para indicar que camino debería seguir en cada caso la ejecución de los procesos.

5.2. Herramientas

Las herramientas que se han usado en el desarrollo del proyecto son las siguientes:

- Lenguaje de programación: Java.
- Herramientas de instrumentación de código: Javassist.
- Gestor de Base de datos: MySQL.
- Plataforma de desarrollo y pruebas: Eclipse SDK

El sistema operativo donde se han instalado las herramientas descritas ha sido Windows Vista.

5.2.1. Java

Java es un lenguaje de programación orientado a objetos.

Las aplicaciones que hemos tenido en cuenta y que hemos usado en las pruebas del desarrollo están desarrolladas en este lenguaje. La solución desarrollada para obtener las trazas también se ha desarrollado en este lenguaje. El código es dependiente de la versión, en este proyecto se ha utilizado la versión 1.5.

De entre las diferentes librerías que ofrece la jdk (el kit de desarrollo de aplicaciones de java) se probado la API Reflection de java para desarrollar una solución al problema propuesto.

En esta versión (1.5) se ha añadido un paquete específico para la instrumentación de código java (java.lang.instrument). Este paquete no se ha probado porque las modificaciones sobre los bytecodes se han de realizar usando otras librerías existentes de instrumentación de código. Con este paquete al cargar una clase en la JVM se obtiene un conjunto de datos sobre la clase (nombre, buffer de datos con los bytecodes, ...), sobre la que hay que hacer las modificaciones que se quieran y devolver la nueva clase (en realidad un array de bytecode) que será la que se use en la aplicación.

5.2.2. Javassist

Javassist (JAVA programming ASSISTant) es una librería de instrumentación (manipulación) de bytecode (código compilado) Java. Permite a los desarrolladores añadir nuevos métodos a clases compiladas, modificar los métodos existentes, añadir variables, etc...

El contenido del software, Javassist, está sujeto a la Mozilla Public License Version 1.1 (the "License").

La API se compone de dos partes: una parte de alto nivel y una de bajo nivel. En la de alto nivel las modificaciones a realizar sobre las clases, métodos y campos se pueden indicar en forma de texto con lenguaje Java. Javassist compila este texto con un compilador propio antes de añadirlo al código original. En este caso el rango de instrucciones que se pueden indicar es limitado. En la parte de bajo nivel las modificaciones se han de indicar directamente en bytecode Java. Por tanto para usar alguna de las capacidades de Java que no están en el rango se ha de usar la parte de la API de bajo nivel. Esta parte no ha sido usada en el desarrollo de este proyecto.

Gracias a la parte de alto nivel, al contrario que otras librerías existentes, Javassist permite realizar todas las modificaciones sobre el código original si tener conocimiento de bytecode Java o de la estructura de un fichero .class (java compilado). Esta es una de las razones por las que ha sido seleccionada esta API para realizar el desarrollo.

La ingeniería se puede realizar tanto en tiempo de compilación como en tiempo de ejecución. En tiempo de compilación se modifica la aplicación antes de la ejecución. La aplicación compilada se carga, se modifica mediante las opciones de la librería Javassist y se almacena una aplicación modificada que será la que se ejecute en lugar de la original. En en tiempo de ejecución se

modifica la clase en el momento de cargar las clases (Javassist también tiene la implementación de un Cargador de Clases propio para poder realizar los cambios en tiempo de ejecución de las clases que se van cargando). En este caso se ha de invocar la aplicación mediante una clase intermedia que será la que haga la carga y transformación de clases antes de llamar a la aplicación que se ha de ejecutar con las clases modificadas. En este caso, las modificaciones no son persistentes, se generan para la ejecución y después el código se desestima, quedando solamente la aplicación original.

La API de Javassist es similar a la API Reflection de Java en algunos aspectos. Javassist permite ver la estructura de una clase java antes de ser cargada por el ClassLoader (cargador de clases de Java). Incluye CtClass, CtMethod y CtField que representan, respectivamente, una clase, un método y un campo que aun no han sido cargados.

La API Javassist, al contrario que la API Reflection, no provee métodos para la creación de nuevas instancias, para la invocación de métodos o para acceder al valor de un campo, ya que los objetos de Javassist (CtClass, CtMethod y CtField) representan clases, métodos y campos que aun no han sido cargados (por tanto aun no están en ejecución). Javassist lo que ofrece son métodos para modificar la definición de las clases.

Para la modificación del código, Javassist tiene un conjunto de variables especiales (que empiezan por $) que se pueden incluir en el código que se modifica y que representan variables del código original. Sin embargo, el código modificado mediante Javassist que hace uso de estas variables especiales necesita el paquete javassist.runtime en tiempo de ejecución. Si estas variables no se usan, la aplicación no encestará ningún paquete adicional de Javasssit en tiempo de ejecución.

El uso de Javassist introduce una penalización en el performance del código original, sin embargo estas son pequeñas gracias al compilador que la librería incluye. El mayor origen de las penalizaciones en el performance son las variables especiales. En cualquier caso, el compilador incluido produce bytecode optimizado para minimizar la pérdida de performance. También, la penalización será dependiente del código que se añada en el proceso de la instrumentación del código.

5.2.2.1. Habilidades de Javassist usadas en el desarrollo

Las habilidades de Javassist usadas en el desarrollo son las siguientes:

- Capacidad para cargar una clase a partir de su nombre completo: indicando el nombre de la clase y el paquete al que pertenece, javassist la carga y la instrumenta antes de que se ejecute. Los nombres de las clases que se van instrumentando los va dando el cargador de clases, por tanto es sencillo realizar modificaciones sobre estas clases cuando se van usando. El cargado de clases de Javassist no carga clases del sistema (ya que estas clases solo se cargan con el cargador de la JVM) por lo tanto en los casos que es necesario instrumentar estas clases hay que buscar métodos alternativos.

- Capacidad para añadir un método a una clase: se puede añadir cualquier método a la clase y una vez instrumentada se podrá invocar el método como el resto de los que hubiese en la clase original.
- Capacidad para cambiar el nombre de un método: se puede cambiar el nombre de un método de una clase. Al cambiar el nombre dejaría de existir el método con el nombre antiguo. Las llamadas se deberían hacer con el nuevo nombre.
- Capacidad para cambiar la implementación de un método: Se puede añadir bloques de código al inicio y al fin de un método. En estos bloques de código se pueden usar los valores de los parámetros del método, lo que no se puede es usar variables locales definidos en el método.
- Capacidad para detectar llamadas a un método de una clase concreta en otra clase: Es posible detectar las llamadas a un método solamente indicando el nombre. Una vez detectada la llamada, la librería ofrece tanto el nombre como los parámetros del método y la clase a la que pertenece.
- Capacidad para obtener las interfaces que implementa una clase: es posible obtener el listado de interfaces que implementa una clase cargada por Javassist.
- Capacidad para obtener las clases de las que extiende una clase: al igual que con las interfaces, es posible obtener el listado de clases de las que extiende una clase.
- Capacidad para obtener los modificadores de clases, métodos y variables: esto permite saber si una clase es pública, privada o abstracta.

Todas estas habilidades han sido usadas durante el desarrollo para obtener una solución al problema inicial.

5.2.3. MySQL

MySQL es un sistema de gestión de bases de datos SQL Open Source (es decir que es posible para cualquiera usar y modificar su código) propiedad de MySQL AB, quien lo desarrolla, distribuye y da soporte.

El software MySQL usa la licencia GPL (GNU General Public License).

En el desarrollo del proyecto se ha usado la versión 5.0 del software.

Para conectar nuestras aplicaciones de prueba con MySQL se ha usado un conector MySQL. Un conector MySQL es un controlador (driver) que proporciona conectividad a una aplicación cliente con el servidor MySQL. Actualmente existen 5 conectores MySQL, de los cuales se ha seleccionado *Connector/J* (que es para aplicaciones desarrolladas en Java).

Cuadro 5.1: Tabla de tipos de datos

Tipo de dato MySQL origen	Tipos de dato Java destino
CHAR, VARCHAR, BLOB, TEXT, ENUM, SET	java.lang.String, java.io.InputStream, java.io.Reader, java.sql.Blob, java.sql.Clob
FLOAT, REAL, DOUBLE PRECISION, NUMERIC, DECIMAL, TINYINT, SMALLINT, MEDIUMINT, INTEGER, BIGINT	java.lang.String, java.lang.Short,java.lang.Integer, java.lang.Long, java.lang.Double, java.math.BigDecimal
DATE, TIME, DATETIME, TIMESTAMP	java.lang.String, java.sql.Date, java.sql.Timestamp
BIT(1)	java.lang.Boolean

5.2.3.1. Connector/J

Este conector MySQL proporciona soporte de controlador para conectar MySQL desde una aplicación Java usando la API de Conectividad con la Base de Datos Java estándar (JDBC).

La versión utilizada de este conector ha sido la 5.1.6.

El conector gestiona las conversiones entre los tipos de dato MySQL y los tipos de datos Java.

En general, cualquier tipo de dato MySQL puede ser convertido al tipo Java java.lang String, y cualquier tipo numérico de dato puede ser convertido a cualquier tipo numérico de Java (aunque puede haber perdida de precisión u overflow si el tipo de dato numérico de Java no es el adecuado).

Las conversiones de datos que están garantizadas por el conector son las indicadas en la tabla 5.1.

En nuestro caso usaremos las siguientes conversiones, aunque a los datos primitivos:

- java.lang.String: java.lang.String
- java.lang.Short: short
- java.lang.Integer: int
- java.lang.Long: long
- java.lang.Double: double
- java.lang.Boolean: boolean

Con este conector, se ha podido probar el almacenamiento de los tipos de datos indicados. Para ello se han creado columnas con los siguientes tipos:

- VARCHAR

- INTEGER
- CHAR
- DOUBLE
- BOOLEAN

De todas las funcionalidades posibles que ofrece esta herramienta, se ha usado para las pruebas una conexión a base de datos local (la aplicación y la base de datos están en el mismo sistema), la ejecución de una consulta (mediante un objeto Statement de Java) y la obtención de los datos (mediante un objeto ResultSet de Java). Hay otras funcionalidades, como uso de Stored Procedures que no se han probado.

Un ejemplo de la conexión usada sería la siguiente:

```
//Carga del driver
Class.forName("com.mysql.jdbc.Driver");
//url de la base de datos
String url = "jdbc:mysql://localhost:3306/bddeva";
//Conexión con la url de la base de datos, el usuario y pw de la misma
Connection con = DriverManager.getConnection(url,"usuario", "password");
Statement stmt = null;
ResultSet rs = null;
try {
stmt = conn.createStatement();
rs = stmt.executeQuery("query");
while(rs.next()){
//Tratamiento de los datos del resultset (resultado de la consulta)
}
} finally {
//liberación de recursos
}
```

5.2.4. Eclipse SDK

Eclipse es la herramienta usada para el desarrollo del código y la ejecución de las pruebas del presente documento (Version: 3.4.0). Con esta herramienta se pueden desarrollar aplicaciones en Java y probarlas. Es software libre. Se le pueden añadir un conjunto de plug-ins que realizan diferentes acciones. En este caso no se ha usado ningún plug-in.

5.3. Desarrollo

El desarrollo del proyecto se dividió en dos partes diferenciadas.

Por un lado se necesitaba una infraestructura con la cual se pudiesen guardar las trazas que se iban obteniendo (los datos de entrada a las aplicaciones) en un fichero plano de texto y por otro se necesitaba la infraestructura con la que

modificar las clases y añadir el bloque de código necesario en cada método (en función de si se almacenan las trazas en el fichero en la ejecución del proceso o de si se obtienen los valores del fichero para la reejecución del proceso).

De la primera parte del desarrollo se obtuvo una librería de clases (Logger), que se ha empaquetado en un jar (logger.jar), con la que obtener las trazas y poder leerlas luego en el momento de la reejecución.

La segunda parte se divide, a su vez, en dos partes diferenciadas, por un lado la modificación de las clases en la ejecución (obtención de trazas) y por otro la modificación en la reejecución (ejecución de la aplicación a partir de los datos almacenados en el fichero de trazas). De esta segunda parte del desarrollo se obtuvo la infraestructura necesaria para la modificación de clases tanto en la ejecución como en la ejecución. Para el desarrollo de esta segunda parte se usó la librería desarrollada en la primera.

5.4. Logger

El logger es la infraestructura creada para realizar tanto el almacenamiento de los datos en el fichero de trazas en la ejecución como su lectura en la reejecución. Se ha desarrollado en Java para facilitar la integración con el resto del código y porque buscamos una solución para aplicaciones desarrolladas en Java.

5.4.1. Estructura

Para el desarrollo de este componente se ha usado el patrón Singleton. El objetivo de este patrón es controlar la creación de objetos, limitando la creación a un número determinado. En nuestro caso, limitamos la creación a una instancia del objeto. Como, a priori, no se sabe cuantos métodos se van a instrumentar, se limita la creación de objetos a solo uno para no crear un objeto por cada método o clase que se instrumente. De esta forma se consumen menos recursos.

El Logger consiste en una clase Java con la estructura indicada en la 5.2.

Se compone de los métodos que obtienen la instancia (getInstance()), el método que lee los datos del fichero (leer()) y los métodos de escritura del fichero (escribir()). El método leer() siempre devuelve un dato de tipo String, ya que se almacena en el fichero de esta forma y al leerlo, a priori, no sabemos de que tipo va a ser. Los métodos escribir() admiten como parámetro cualquier tipo primitivo de Java o un valor de tipo String, que son los datos que hemos tenido en cuenta.

El mismo objeto lleva a cabo la lectura y la escritura de los datos. En la creación de una instancia del objeto se indica el modo en que se abre el fichero: LECTURA o ESCRITURA en función de si se leen o se escriben las trazas respectivamente.

En la creación del objeto se necesitan dos datos:

- El *path* donde se va a dejar el fichero de trazas. Hay que asegurar que el proceso que es esta ejecutando tiene permisos para dejar el fichero en el path indicado, si no es así no se podrán recuperar las trazas de ningún

Cuadro 5.2: Estructura de la clase Logger.

Logger
+MODO_LECTURA
+MODO_ESCRITURA
+getInstance()
+getInstance(String)
+getInstance(String,String)
+escribir(Object)
+escribir(boolean)
+escribir(short)
+escribir(long)
+escribir(float)
+escribir(double)
+escribir(char)
+escribir(byte)
+escribir(int)
+escribir(char [])
+escribir(byte [])
+leer()

modo (ya que no se guardarán). El path incluye, además del directorio donde se va a dejar el fichero creado, el nombre que se le va a dar al fichero. El nombre del fichero es, ahora mismo, siempre el mismo. Sin embargo en una misma máquina podrían estar ejecutándose multitud de procesos distintos que deberían dejar su fichero de trazas propio y además saber distinguirlo de los ficheros que crean el resto de procesos. Este problema está aun por resolver.

- El *modo* en que se usa el fichero (escritura o lectura) en función de si es ejecución o reejecución respectivamente. En la ejecución, que se considera la ejecución original en la máquina remota, se obtienen los datos y se almacenan en el fichero de trazas. En la reejecución, que se considera la ejecución de la aplicación a partir del programa y el fichero de datos en la máquina local, se obtienen los datos almacenados en el fichero de trazas para llevar a cabo la ejecución y comprobación de que el programa se ha ejecutado correctamente (es decir que no se ha modificado su código deliberadamente).

Para indicar estos datos existe un fichero de configuración en el path */resources*, que se llama *logger.properties*, que tiene los dos pares de valores. El fichero tiene la siguiente estructura:

```
#Path del fichero de trazas
path=trazas.dat
#Modo en que se usa el logger. Hay dos modos definidos:
#w: es el modo de escritura en el que se almacenan las trazas
```

Cuadro 5.3: Tabla de Wrappers de tipos primitivos de Java

Tipo primitivo	Clase wrapper
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
boolean	java.lang.Boolean
char	java.lang.Character

#r: es el modo de lectura en el que se obtienen los datos necesarios para la reejecución de la aplicación
modo=w

Para indicar el modo, independientemente de lo que dice el fichero de configuración, se puede usar el método getInstance(String), donde se pasa el modo (MODO_LECTURA (r) o MODO_ESCRITURA(w)), en función de como se quiera abrir el fichero.

El orden asignado a las trazas en el caso de escritura del fichero será el orden en que aparezcan los datos en la aplicación. De esta forma con leer el fichero de forma secuencial (la forma habitual de leer los ficheros de datos con Java) en la reejecución, las trazas que se vayan leyendo contendrán los valores que se deberán ir aplicando a las variables para obtener el resultado.

No se podrán leer datos por separado. Ni escribir datos con datos no secuenciales. Tanto en la lectura como en la escritura, será el objeto internamente el que decida que se lee y que número se asigna a la traza en cada caso.

5.4.2. Ejecución: almacenamiento de trazas

Para almacenar un valor será suficiente con obtener la instancia (que se creará si no existe y se reutilizará si existe) y llamar al método escribir, al que se le pasa un parámetro, de tipo Object o del tipo primitivo del que sea el dato, con el valor a almacenar. Si el parámetro que se pasa es de tipo Object habrá que convertir cada uno de los datos de tipo primitivo a la clase Java que lo envuelve.

A parte existe un método escribir por cada uno de los tipos primitivos de Java, que almacenan en el fichero el valor que se pasa por parámetro. El método escribir es capaz de reconocer las clases que envuelven a los tipos primitivos para almacenar de forma correcta el valor en el fichero.

Las correspondencias de tipos primitivos y clases wrapper (que los envuelven) son los mostrados en la 5.3.

5.4.3. Reejecución: obtención de trazas

En el caso de reejecución, para obtener el valor de una variable, será necesario obtener la instancia (indicando el modo adecuado) y llamar al método *leer()*.

Este método, al ser llamado por primera vez, lee el fichero completo y monta una estructura de tabla con los identificadores (número de orden) y los valores asociados a cada uno de ellos (que pueden ser un valor en el caso de tipos simples o un conjunto de valores en el caso de un array). Internamente mantiene un contador de las veces que se va llamando para devolver el valor de la variable con el número de orden que toca.

En este caso, al provenir de un fichero de texto no se puede saber a priori cual era el tipo original de la variable. Por eso todos los valores se devuelven en forma de String o array de Strings y es competencia del programador convertir el String leído al tipo de dato necesario para la aplicación en el momento de la instrumentación.

5.4.4. Fichero de trazas

El fichero resultado de la ejecución se compondrá de un listado de pares de valores:

- Un identificador de la traza: en nuestro caso será un numérico que indicara el orden de como vayan leyéndose los datos de entrada de la aplicación. El número asignado no se usará para identificar la variable con la que hay que asociarlo en la reejecución ya que el orden de las variables no cambia de la ejecución a la reejecución. En el momento de la reejecución se leerá el conjunto completo de trazas y se ordenará mediante este numérico. Después se irán asignando los valores con el orden indicado.

- El valor de entrada: el valor asignado será siempre de tipo String, ya que tratamos con un fichero de texto. En el momento de la reejecución deberemos trasladar el valor leído en forma de String al tipo de dato que necesite la aplicación para funcionar correctamente.

Con esta estructura de fichero (identificador único + valor de entrada) se comprobó que no era seguro almacenar todos los tipos de variables que Java proporciona. En el caso de intentar almacenar un Array, donde no sabemos que número de elementos hay (pueden ser infinitos) si se almacenaban en una sola línea se tenía un problema al separar los elementos, ya que no se sabía donde empezaba cada uno de ellos y si se almacenaban en diferentes líneas con un identificador único para cada una de ellas teníamos el mismo problema (no se sabia cuantos elementos formaban parte del array). Para solucionar esto, se optó por asignar el mismo identificador a los valores que pertenecían a una variable de tipo Array, de esta forma, al leer un conjunto de líneas con el mismo identificador era sencillo convertirlo al Array con tipo de datos necesario para la correcta reejecución de la aplicación.

Cuadro 5.4: Ejemplo de fichero de trazas.

Identificador	Valor	Tipo de dato
0	val 1	Array de 4 posiciones
0	val 2	
0	val 3	
0	val 4	
1	val 5	Valor simple
2	val 6	Valor simple
3	val 7	Array de 3 posiciones
3	val 8	
4	val 9	Valor simple
...	...	...

Cuadro 5.5: Ejemplo de fichero de trazas con datos

Identificador	Valor	Tipo de dato
0	51	Array de 4 enteros
0	462	
0	516	
0	174	
1	Eva	String
2	5	int
3	12345678J	Array de 2 String
3	07721443M	
4	30	int
...	...	...

En la tabla 5.4 se muestra un ejemplo de como sería un fichero de trazas. En la tabla 5.5 se muestra, para el fichero de trazas de la tabla 5.4, un ejemplo de fichero con valores reales.

5.4.5. Comprobación del fichero de trazas

El objetivo del fichero de trazas generado en la ejecución es el de poder reejecutar la aplicación en el caso en el que se duda de que el host que ha ejecutado el agente ha realizado algún ataque sobre el mismo que puede provocar que los resultados de la ejecución obtenidos no sean los correctos (los correctos tal y como se ha programado el agente y como debería haber sido ejecutado).

En el momento de la reejecución tendremos el código original del agente y el fichero de trazas del que obtener los valores necesarios para la ejecución. Para comprobar que la ejecución ha sido correcta basta con ejecutar la aplicación con el fichero de trazas y comprobar los resultados obtenidos, tanto del host remoto como el obtenido en nuestro local con las trazas enviadas por el host remoto.

Debido a que la ejecución de un agente puede ser costosa, en función de las

acciones que realice, se pueden hacer una serie de comprobaciones previas que pueden prever, sin realizar la ejecución, que la ejecución en el host remoto no ha sido la correcta.

Las acciones a realizar son las siguientes:

- Comprobación del número de variables almacenadas en el fichero.
- Comprobación de los tipos de datos de las variables almacenadas en el fichero.

5.4.5.1. Número de variables almacenados en el fichero

Una verificación rápida para saber si un fichero puede ser sospechoso de ser incorrecto sería la comprobación de que el número de variables es el mismo que los datos de entrada de la aplicación.

De la misma forma que la aplicación es instrumentada para obtener los datos de entrada, se puede instrumentar para obtener el número total de datos que deberíamos encontrar en el fichero. Comparando este número con el número de datos (número de variables distintos) almacenados en el fichero enviado por el host remoto, si no coinciden se puede dar la ejecución del host remoto como errónea. En este caso habrá algún segmento de código del agente que haya sido modificado o que se haya saltado en la ejecución y podría ser para que el host remoto obtuviese algún beneficio.

Tal y como se ha desarrollado el logger, podríamos saber cuantos datos de entrada distintos hay en la aplicación. Sin embargo tendríamos un problema con los arrays, ya que no sabríamos si realmente son correctos al poder tener un número variable de valores dentro. En el caso de arrays se guarda en el fichero un dato con el mismo identificador para cada uno de los valores contenidos dentro. Esto nos dice que valores pertenecen a un array o colección, pero no nos dice cuantos valores se han de encontrar en el fichero para cada uno de los arrays. Para poder comprobar el número de valores que hay que encontrar en cada uno se debería modificar la estructura del fichero de trazas que se está almacenando.

El hecho de pasar esta comprobación no asegura que el fichero sea realmente correcto, para ello se deberán comprobar los tipos de datos o reejecutar el código con los valores del fichero.

5.4.5.2. Tipos de datos almacenados en el fichero

Igual que en el caso anterior, se puede instrumentar la aplicación para comprobar los tipos de los datos del fichero. Esta comprobación es menos costosa que la ejecución real del agente y puede destapar al host remoto si alguno de los datos no coincide en tipo con los del fichero de trazas.

El hecho de que los tipos de datos coincidan no es indicativo de que el agente se ha ejecutado de forma correcta y si esta comprobación no indica ningún error se debe reejecutar el agente para comparar los resultados de las dos ejecuciones.

5.5. Ejecución para la obtención de trazas y reejecución a partir del fichero de trazas del host de ejecución

En este apartado se describen las acciones llevadas a cabo para el desarrollo de los componentes que permiten obtener las trazas del programa en un fichero de datos durante la ejecución del agente en un host de su itinerario, así como el desarrollo de los componentes que permiten volver a ejecutar la aplicación original partiendo de los valores almacenados en un fichero de datos que se supone es el resultado de la ejecución del agente en cualquier host.

Esta parte del desarrollo se divide a su vez en tres partes diferenciadas, en función del origen de los datos a almacenar en el fichero de trazas:

- Obtención de trazas de los datos de entrada de la aplicación: los datos de entrada en la ejecución del agente si los hubiera
- Obtención de trazas de los datos obtenidos al acceder a ficheros: datos obtenidos de ficheros ubicados en hosts donde el agente se ejecuta.
- Obtención de trazas de los datos obtenidos al acceder a bases de datos: datos obtenidos de las bases de datos, tanto locales como remotas, usadas por los hosts de la red durante la ejecución del agente.

A continuación se describe cada uno de los casos.

5.5.1. Datos de entrada de la aplicación

En Java los procesos standalone entran siempre por el mismo método.
El método tiene la siguiente firma:
public static void main(String [] args);

Como datos de entrada tiene un array de Strings que puede ser tan largo como se quiera, incluso no tener ningún valor (en ese caso no habría que guardar ningún dato). Así, los parámetros de entrada de un proceso Java son siempre Strings y es competencia del programador convertir los datos de entrada al tipo de dato que la aplicación necesite. Por esta razón no tendremos que tener en cuenta más tipos de datos, ya que suponemos que el programador realizará la conversión al tipo que necesita de forma correcta.

Para el desarrollo de la solución se han usado dos soluciones distintas, una con la API Reflection y otra con Javassist. La primera idea fue hacer el desarrollo con la API Reflection de Java, sin embargo es una API que penaliza mucho el rendimiento de la aplicación modificada. Esta API no está indicada para ser usada en métodos o clases que se usan mucho, en el caso de los parámetros de entrada penaliza menos porque por ahí solo pasa una vez, pero para el resto de soluciones no era una buena idea usarla. Este es el único problema que se solucionó con esta API.

5.5.1.1. Reflection

Una de las habilidades de Reflection es la de inspeccionar una clase para obtener los métodos que tiene definidos y poder hacer la llamada a los métodos encontrados. Además de encontrar los métodos definidos, con Reflection es posible saber que parámetros se le pasan y recoger sus valores.

Sabiendo que el método de entrada es siempre el "main", es sencillo buscar ese método en la clase y hacer la llamada para ejecutar la aplicación.

Por tanto para solucionar la obtención de trazas se desarrolló una clase que mediante Reflection inspecciona la clase, obtiene el método main y justo antes de hacer la invocación del método encontrado almacena los valores de los parámetros en el fichero de trazas.

En el caso de la reejecución el sistema es el mismo, pero en lugar de obtener los valores de los parámetros de entrada y almacenarlos en el fichero de trazas, lo que se hace es obtener los valores del fichero de trazas y usarlos para pasárselos como parámetros al método main al invocarlo.

La desventaja de este método, a parte de la pérdida de performance como se ha comentado anteriormente, es que se ha de realizar la llamada a aplicación a través de una clase intermedia que es la que hace la llamada real a la clase que se quiere ejecutar. De esta forma debería haber algún mecanismo que al intentar arrancar una aplicación, llamase en su lugar a la clase intermedia que inspecciona la clase que realmente se quiere ejecutar.

La ventaja de este método es que no necesita referencia a librerías que no van en la jre de Java y por tanto no va a tener problemas de que no encuentra clases necesarias para la ejecución.

5.5.1.2. Javassist

Al contrario que con la API Reflection que inspecciona las clases en ejecución, Javassist es capaz de modificar clases antes de la ejecución. Por lo tanto con Javassist no es posible obtener los valores de las variables, se modifican las clases en el momento de cargarlas y al usarlas en la aplicación se usan las modificadas. En la modificación se añade a la clase un bloque de código que almacena los valores de las variables.

Para solucionar el problema con Javassist se usaron las siguientes características de la librería:

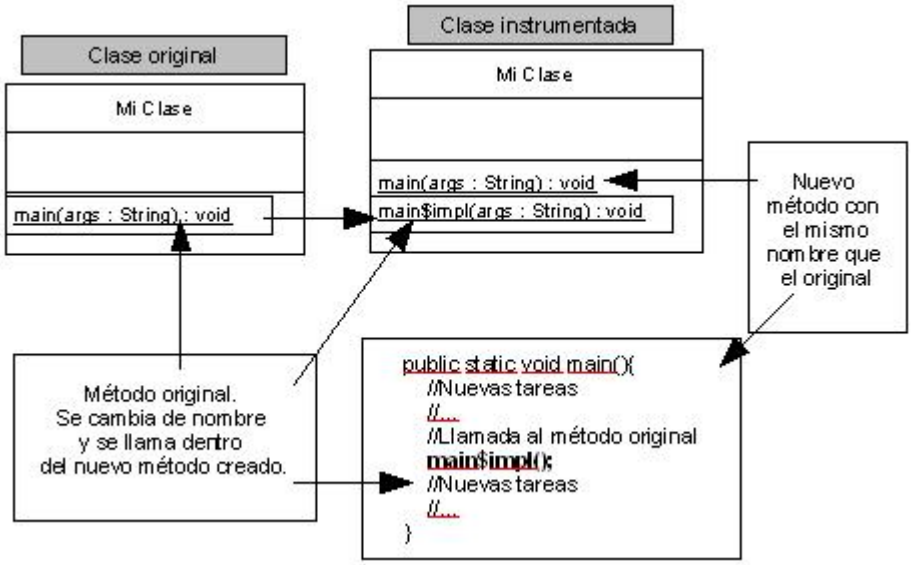
- Habilidad para crear nuevos métodos en una clase
- Habilidad para renombrar métodos ya existentes en la clase

Por un lado renombramos el método main original de la clase y por otro creamos uno nuevo que llamaba al método original (el cambiado de nombre) y además almacenaba los datos de entrada de la aplicación.

El esquema de la solución sería el indicado en la figura 5.1.

Al ejecutar la aplicación con la clase instrumentada, el método llamado seria main (el nuevo que hemos creado que se llama como el original) que realizaría las acciones originales (el método original cambiado de nombre, es decir llamar

Figura 5.1: Instrumentación para los datos de entrada a la aplicación



a main$impl()) más las nuevas que definimos (almacenamiento de los valores en el fichero de trazas en el caso de la reejecución y obtención de los datos del fichero en el caso de la reejecución).

La ventaja de este método es que la pérdida de performance es menor que con Reflection. Sin embargo, este método necesita la librería de Javassist runtime para poder ejecutarse. La necesita porque en la instrumentación se usan las variables especiales de Javassist (empiezan por $) para obtener los valores de los parámetros que se pasan al método (que son los que hay que almacenar en el fichero).

5.5.2. Acceso a base de datos

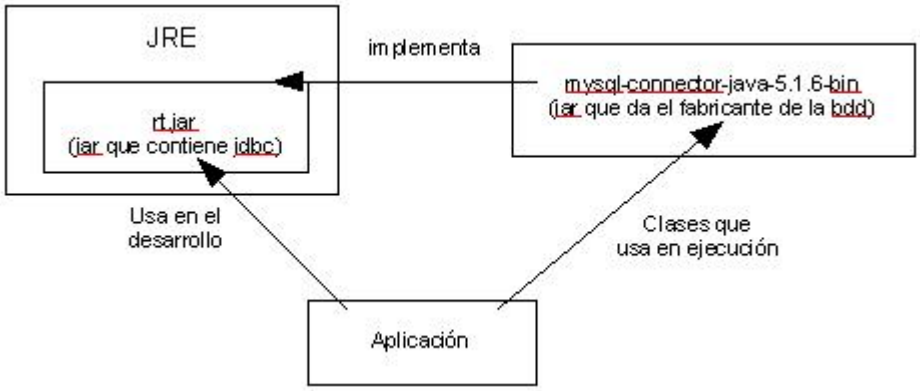
Para el acceso a bases de datos, Java tiene la API JDBC. Con esta API se puede acceder a cualquier tipo de datos tabulados, especialmente almacenados en bases de datos relacionales.

JDBC da soporte para realizar las siguientes acciones necesarias en un acceso a base de datos:

1. Conexión al origen de datos
2. Envío de querys y updates hacia la base de datos
3. Obtención y proceso del resultado obtenido de la base de datos como respuesta a la query enviada.

En el caso de la ejecución supondremos que la base de datos existe y que la conexión se puede realizar de forma correcta. Así no habrá problemas con los

Figura 5.2: Esquema del uso de librerías JDBC



pasos 1 y 2 y solo tendremos en cuenta el paso 3. En resumen, el objetivo en la ejecución será el registro en el fichero de trazas de los datos obtenidos de la base de datos como respuesta a las querys que se le envían en la aplicación.

En el caso de la reejecución no podemos suponer que la base de datos exista ni que la conexión se vaya a realizar de forma correcta. De esta forma se deberá implementar algún mecanismo para inhibir las acciones de los pasos 1 y 2 pero permitiendo que la aplicación llegue al paso 3 (en realidad se debería saber que ha pasado en el paso 1, ya que en la ejecución original podría haber algún tipo de problema de conexión, de que la base de datos no esté disponible, etc..). En el paso 3 usaría los datos almacenados en el fichero de trazas para la ejecución de la aplicación.

A continuación se detallan los pasos seguidos para el desarrollo tanto en la ejecución como en la reejecución.

5.5.2.1. Ejecución

Como se ha comentado, en el caso de la ejecución se tendrán que almacenar en el fichero de trazas los datos que se obtienen de la base de datos.

JDBC se basa en interfaces que definen que comportamiento han de tener las clases que realizan el acceso real a las bbdd. Las clases que implementan las interfaces dadas son desarrolladas por el propietario de cada sistema gestor de bases de datos y dependen de la base de datos que se usa. En el esquema de la figura 5.2 se muestra las relaciones entre clases.

La interfaz a partir de la cual se obtienen los datos de la base de datos es *java.sql.ResultSet*. Esta interfaz declara métodos para obtener los valores de las columnas de las tablas de la base de datos..

Los métodos (que tratan datos primitivos de Java) son los siguientes:

- boolean **getBoolean**(int columnIndex)
- boolean **getBoolean**(String columnName)

- byte **getByte**(int columnIndex)
- byte **getByte**(String columnName)
- byte[] **getBytes**(int columnIndex)
- byte[] **getBytes**(String columnName)
- int **getConcurrency**()
- String **getCursorName**()
- double **getDouble**(int columnIndex)
- double **getDouble**(String columnName)
- int **getFetchDirection**()
- int **getFetchSize**()
- float **getFloat**(int columnIndex)
- float **getFloat**(String columnName)
- int **getInt**(int columnIndex)
- int **getInt**(String columnName)
- long **getLong**(int columnIndex)
- long **getLong**(String columnName)
- int **getRow**()
- short **getShort**(int columnIndex)
- short **getShort**(String columnName)
- int **getType**()

Todos estos métodos debían ser instrumentados para recoger los valores en el fichero de trazas.

La primera idea fue instrumentar la interfaz. Sin embargo, la interfaz es solo un contrato donde se definen la firma de los métodos que han de implementarse en una clase. Los métodos declarados en una interfaz no están definidos (no tienen código con lo que deben hacer). Al no tener definición, los métodos de la interfaz no se pueden instrumentar.

La segunda idea fue implementar directamente la clase que hace el acceso a la base de datos. Esta clase (que es la desarrollada por cada fabricante) ha de implementar la interfaz java.sql.ResultSet (es la que define los métodos de acceso), por tanto los métodos a instrumentar serían los mismos que los definidos en la interfaz. Como hemos comentado, la clase que realmente hace el acceso a base de datos es del fabricante del sistema gestor de base de datos, por tanto

la instrumentación hecha para un fabricante no funcionaría para otro. A priori no se puede saber que base de datos se va a usar, por lo tanto se necesita una solución genérica. La idea, para resolver este problema, fue buscar que clases implementaban la interfaz (java.sql.ResultSet). De esa forma tendríamos la clase donde realmente se implementa el código. Esa clase, si se encontrase, si que se podría instrumentar. Además, automatizando la búsqueda la solución seria independiente del fabricante del gestor de base de datos usado. Javassist permite conocer que interfaces implementa una clase cargada, por tanto no es complicado saber si una clase dada implementa una interfaz o no. El problema fue un poco más complicado que buscar solamente las clases que implementan la interfaz, ya que hay una jerarquía de clases abstractas hasta llegar a la que realmente hace el acceso a los datos. La solución, para encontrar la clase buscada, necesitó un bucle donde se van buscando las clases que implementan o extienden de las interfaces y clases intermedias que se van encontrando. Una vez encontrada la clase que buscamos (la que primera que implementa la interfaz o extiende de una clase abstracta que implementa la interfaz y que no es abstracta) se debían instrumentar los métodos para que, además de hacer la llamada habitual (obtener el valor de la base de datos), guardase el valor de la variable en el fichero de trazas.

Para implementar esta solución se usaron las siguientes habilidades de Javassist:

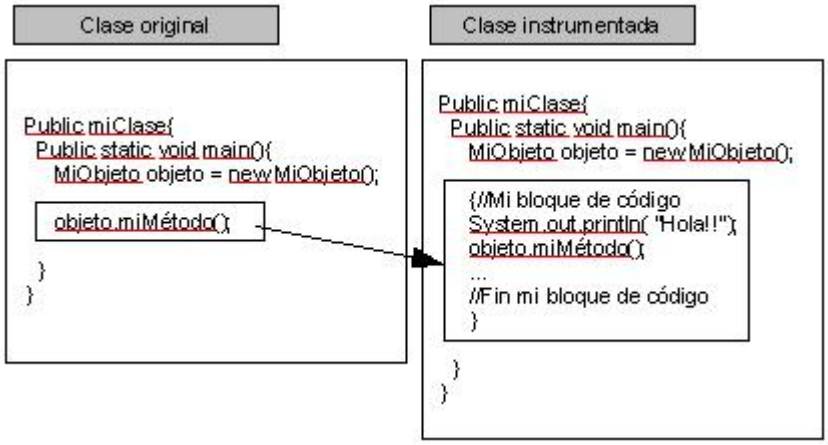
- creación de nuevos métodos
- cambio de nombre de un método
- Obtención de clases que implementan una interfaz dada
- Obtención de clases que extienden de una clase dada

La solución sería la misma que la adoptada para los parámetros de entrada (la solución con Javassist). Así cambiando el nombre al método original y creando un método nuevo con el nombre del método original se pudo solucionar el problema. El nuevo método tendría dentro nuestra llamada a un método que almacena el dato en el fichero de trazas y la llamada al método original (que ahora tiene un nombre nuevo).

Con esta solución es posible registrar los valores en el fichero de trazas. Sin embargo se almacenan todas las llamadas a la base de datos. Los métodos definidos por la interfaz java.sql.ResultSet se usan, además de para obtener los valores de la base de datos, para otras operaciones internas como obtener datos de configuración y otros datos necesarios en el funcionamiento de la clase que provee el fabricante del gestor de base de datos. Con esto se almacenan valores que no son realmente entradas de la aplicación y que además son dependientes de la base de datos que se está usando. Por esta razón fue necesario buscar otro tipo de solución.

La tercera idea, para solucionar el problema surgido en la última, fue utilizar otra de las habilidades de Javassist que es la de detectar las llamadas a un método de una clase determinada. De esta forma no se modifica la clase que

Figura 5.3: Instrumentación de la llamada a un método



accede a la base de datos (las clases de sistema, ni las que da el propietario de la bbdd) sino que se modifica la clase que usa la interfaz (la aplicación desarrollada). De esta forma es fácil cambiar la llamada a un método por un bloque de código que incluya la llamada original y el código nuevo que queremos añadir.

La estructura de la solución es la siguiente la que se muestra en la figura

En este caso, detectando todas las llamadas, el problema era el mismo que con la idea anterior, que se guardaban más datos que las entradas directas a la aplicación, ya que se instrumentaban también las clases que realizan el acceso a la base de datos. Para eliminar el problema se limitó a que las clases donde se detectaban las llamadas no estuviesen dentro del paquete del propietario del gestor de la base de datos (en nuestro caso "com.mysql.jdbc"). Esto no lo podíamos hacer en el caso anterior porque al modificar el método que obtiene el dato directamente, no se podía distinguir en el acceso si se estaba obteniendo un valor necesario para la aplicación o un valor necesario para el funcionamiento interno del conector pero que no era necesario en la aplicación. Con esta nueva solución se puede limitar a que se detecten las llamadas en el conjunto de clases que nos interese y así solo obtener los datos necesarios para la ejecución de la aplicación. La solución que hemos adoptado es dependiente de la base de datos, aunque se podría limitar más los paquetes que no se instrumentarán para que sirva para otras bases de datos.

5.5.2.2. Reejecución

En este caso, además de instrumentar las clases para cambiar la obtención de los datos de la base de datos por la obtención de los datos del fichero se tenían que tener en cuenta el resto de pasos:

- Conexión a base de datos

- Lanzamiento de la query para obtener el resultado

Ya que no podíamos suponer que la base de datos existía ni que se fuese a realizar la conexión de forma correcta. La reejecución deber funcionar a partir del fichero de trazas únicamente.

El caso de la conexión a la base de datos se puede resolver, si se supone que no hay problemas de conexión ni de disponibilidad de la base de datos. Se puede instrumentar el método que realiza la conexión para obtener una conexión de cualquier base de datos que sepamos que está disponible (una local o accesible para el sistema que realiza el proceso de reejecución). Una vez obtenida una conexión cualquiera, con el objeto creado se pueden crear los objetos statement necesarios sin que de un error. Para poder realizar esto se debe saber de antemano que base de datos se está usando y como se está haciendo la conexión.

El lanzamiento de la query se debería inhibir, ya que los datos del resultado se van a obtener del fichero de trazas.

El caso del obtención de los datos del fichero de trazas no era obvio debido a la forma en que se lanza una query y en que se accede a los datos que esta llamada devuelve. La forma es la siguiente (se han obviado las comprobaciones y el control de errores):

```
Statement st = this.con.createStatement();
ResultSet rs = st.executeQuery( "select * from personas" );
while( rs.next() ){
//Realizar acciones con los datos obtenidos
}
```

La documentación de Java indica que la clase ResultSet no tiene ningún método para obtener el total de datos que devuelve la query lanzada. De esta forma no sabemos, ni podemos saber instrumentando el código, cuantos resultados del fichero de trazas debemos tener en cuenta (cuantos forman parte del resultado de la query). Por lo que hemos supuesto al principio de que no dispondremos de la base de datos, necesitamos un mecanismo que nos devuelva un valor en el método next() que es el que nos dice si hay más resultados de la query o no. Para resolver este problema se decidió instrumentar el método next() también en la ejecución para que almacenase el valor y así poder obtener del fichero de trazas cuantas veces debemos iterar el bucle para obtener los datos de la base de datos. Esto se hizo así por que cada valor que se recoge de la base de datos va en una línea nueva del fichero (al ser Strings si se ponen todas en la misma línea del fichero no podríamos saber donde empieza y acaba cada uno de los valores). Con esta solución se pudo reejecutar el proceso obteniendo los valores del fichero correctamente.

5.5.3. Acceso a ficheros

El acceso a ficheros se realiza en Java con la las clases que se encuentran dentro del paquete **java.io**. Este paquete contiene clases que gestionan el tratamiento de datos provenientes de diferentes orígenes (datos provenientes de dispositivos, de otros programas, de arrays de memoria, de ficheros almacenados en

disco, etc...). El desarrollo para cualquiera de los tipos de origen es el mismo. Por simplicidad se ha seleccionado, para el desarrollo y las pruebas, la lectura de datos provenientes de un fichero almacenado en disco.

En Java existen diferentes tipos de clases que realizan accesos a datos en función del tipo de dato que tratan. Los hay que tratan con bytes, con caracteres, con tipos simples de Java, con Objetos...Básicamente se dividen en dos tipos: clases que gestionan streams de datos (un flujo de datos sin estructura) las que gestionan ficheros estructurados (la gestión de ficheros se realiza independientemente de la plataforma).

Para cada tipo de stream de datos de la siguiente lista existen clases específicas que gestionan la lectura/escritura:

- **Streams de bytes:** gestionan datos binarios sin tratar
- **Streams de caracteres:** gestionan datos en forma de carácter, realizando la conversión a y hacia el local character set.
- **Buffered streams:** optimizan las entradas y salidas reduciendo el número de llamadas hechas a la API nativa
- **Data streams:** gestionan entradas binarias de datos primitivos Java y Strings.
- **Object Streams:** gestionan entradas de objetos de tipo Object. Este tipo no ha sido probado ya que está fuera del alcance del proyecto.

Para el tratamiento de ficheros estructurados existen las siguientes clases en el paquete:

- **File:** trata ficheros (creación, borrado, movimiento..., pero no lectura de los mismos)
- **Random Acces File:** Permite acceso no secuencial o aleatorio a los ficheros.

Además de estas clases, debido a la herencia, cualquier desarrollador podría desarrollar una clase propia para realizar la lectura de su origen de datos particular. En el presente desarrollo no tendremos en cuenta estas posibles clases, ya que hemos supuesto que todas las clases usadas en las aplicaciones pertenecen al paquete java.io, distribuido en el jar rt.jar.

Las clases de tipo Stream se basan en dos clases abstractas básicas, de las que extienden el resto de clases. Por ser abstractas no pueden ser instanciadas en una aplicación. Son las siguientes:

- **InputStream:** manejan datos binarios (entradas de bytes de 8 bits). Están indicadas para operaciones de entrada a más bajo nivel únicamente. Los métodos que realizan el acceso a los datos son los siguientes:
  - abstract int **read**(): Lee el siguientes byte de data de stream de entrada. Devuelve un entero con la codificación del byte leído.

  - int **read**(byte[] b): Lee un número de bytes indeterminado del stream de entrada y lo almacena en el array b. Devuelve el número de bytes leídos y en el array los bytes que se han leído realmente.
  - int **read**(byte[] b, int off, int len): Lee hasta len bytes del stream de entrada y lo almacena en el array b. Off indica el número de bytes que salta desde el punto de lectura. Devuelve el número de bytes leídos y en el array los bytes que se han leído realmente.

- **Reader:** Manejan caracteres (16 bits). Los datos son traducidos automáticamente con el juego local de caracteres. Los métodos que realizan el acceso a los datos son los siguientes:

  - int **read**(char[] cbuf): Lee caracteres y los almacena en un array
  - abstract int **read**(char[] cbuf, int off, int len): Lee caracteres y los almacena en una parte de un array.

Todos los métodos indicados anteriormente devuelven un entero para poder discriminar un dato leído en el fin de un fichero, que se representa en estas clases por un **-1**.

A continuación se detallan los pasos seguidos para el desarrollo tanto en la ejecución como en la reejecución.

5.5.3.1. Ejecución

La primera idea fue instrumentar las clases básicas (InputStream, Reader), sin embargo no es posible ya que las clases son abstractas. Esto significa que los métodos (alguno de ellos) que hacen el acceso a los datos no están implementadas, por tanto no se pueden instrumentar. Además, las clases que realmente se usan descienden de estas básicas, pero pueden invalidar los métodos originales creando unos nuevos propios de la nueva clase con el mismo nombre que los antiguos (herencia).

La segunda idea fue, al igual que en el caso del acceso a bases de datos, obtener las clases que descienden de estas básicas, para así instrumentar las que realmente se usan en el código. Esto es factible técnicamente, sin embargo el resultado no es distribuible como aplicación. Las clases pertenecen al Runtime Environment de Java 2 y distribuyendo estas clases modificadas se contraviene a su licencia de código.

Finalmente, la solución adoptada fue la misma que para el acceso a base de datos. Interceptar las llamadas a los métodos read() de las clases de la aplicación para instrumentar la clase que las usa (no son clases de sistema, sino de la aplicación).

Inicialmente se instrumentaron los métodos básicos de InputStream (los métodos read() descritos anteriormente, que son heredados por todas las clases que descienden de ésta). Observando el comportamiento de cada método tenemos lo siguiente:

- int **read**(): Este método devuelve un entero que es el dato leído (la codificación del byte leído) o un -1 en el caso de haber llegado al fin de fichero. Este dato (el entero) es el que debería almacenarse en el fichero de trazas. En este caso fue simple modificar el cuerpo del método para hacer la llamada original al método para obtener el valor leído y añadir después el logger para almacenar en el fichero de trazas el valor leído.

- int **read**(byte[] b): Este método devuelve un entero, pero en este caso no representa el valor leído del fichero. La lectura se almacena en el parámetro de entrada (el array de bytes) y lo que devuelve el método es el número de bytes que son válidos para la lectura (la lectura podría ser de menor longitud que la longitud del array que se pasa por parámetro) o un -1 en el caso de haber llegado al fin de fichero. En este caso había que adoptar otra solución, ya que se debían almacenar los dos datos: el array leído y el entero que devuelve para saber cuales de ellos son válidos en cada lectura.

- int **read**(byte[] b, int off, int len): Este caso es igual que el anterior, ya que la lectura está almacenada en el array de byte de entrada. En este caso también se han de almacenar los dos valores.

Otro tema importante es que los tres métodos a instrumentar se llaman igual (read), por tanto en el momento de interceptar la llamada se necesitaba una forma de poder distinguir cada uno de ellos para realizar el almacenamiento correcto de los datos. La primera idea (por ser la más sencilla) fue diferenciar los métodos por el número de parámetros que estos aceptan. Aunque este no es un método fiable, ya que podrían existir múltiples métodos con el mismo nombre y el mismo número de parámetros, para una primera prueba fue suficiente para diferenciar los métodos e instrumentarlos correctamente. Así, en el caso de no tener parámetros se almacena en el fichero de trazas el valor retornado por el método y en los otros casos se almacenan el valor de retorno y el valor del primer parámetro (el array de bytes leídos). Para probar los métodos de InputStream, con este desarrollo se obtuvieron correctamente los valores en el fichero de trazas. Las pruebas se realizaron con la clase FileInputStream, que desciende directamente de InputStream, se usa para leer datos de ficheros y tiene estos tres métodos implementados. Como se ha indicado anteriormente la clase InputStream es abstracta y por tanto no puede ser instanciada.

Una vez solucionado este problema se siguió con la instrumentación de los métodos de la clase Reader. En este caso tenemos los mismos problemas que con los de la clase InputStream: en los dos métodos los datos leídos del fichero se almacenan en el array de chars que se pasa por parámetro y lo que retorna el método es el número de caracteres que son válidos en la lectura. Para instrumentar estos métodos se procedió a instrumentar las clases igual que con la clase InputStream. Sin embargo, al añadir estos métodos tenemos que hay dos métodos que tienen un parámetro y dos que tienen 3 parámetros. Mirando los métodos se puede ver que la lectura siempre se almacena en el primer parámetro y que siempre se devuelve el total de posiciones leídas válidas. Como se desarrolló el logger lo suficientemente inteligente como para almacenar los datos en

función de su tipo no se hizo necesario diferenciar los métodos de ninguna manera. El logger sabe como almacenar un array de bytes o un array de caracteres en función de lo que se le pasa para escribir en el fichero.

Con este desarrollo se consiguió obtener correctamente los datos en el fichero de trazas para los métodos básicos.

Con el desarrollo hasta este momento sólo se habían hecho pruebas con datos de tipo byte y char. Sin embargo hay clases de acceso a streams que tratan con otros tipos de datos:

- **BufferedReader:** lee cadenas de caracteres (hasta un salto de línea). El método devuelve un null cuando no hay más que leer. Si hay datos para leer devuelve cada línea en un String.
- **DataInputStream:** lee datos primitivos de Java y Strings. Para leer un fichero con esta clase, el fichero a leer se ha de haber generado con la clase DataOutputStream, que escribe datos de tipo primitivo y String. Con esta clase se pudo probar la lectura de casi todos los tipos de datos primitivos.

La clase BufferedReader trata datos de tipo String. Tiene un método readLine() que lee hasta que encuentra en el fichero un retorno de carro. Se intentó instrumentar una aplicación que hacía uso de esta clase. El único problema que se encontró es que se podía leer un valor nulo y que se necesitaba una forma de almacenar este valor en el fichero de trazas. Esto es porque en lugar de un -1 para indicar el fin de fichero este método devuelve un null cuando no hay más datos que leer. El problema se resolvió guardando la cadena de texto "null" en el fichero de trazas. Realmente no es una solución definitiva, ya que "null" es una cadena de texto válida que podría existir en un fichero sin indicar fin de fichero. Por tanto el problema sigue aún sin resolverse definitivamente.

La clase DataInputStream permite leer tipos primitivos de Java. A parte de los métodos read() heredados, tiene los siguientes métodos de lectura:

- boolean **readBoolean**()
- byte **readByte**()
- char **readChar**()
- double **readDouble**()
- float **readFloat**()
- void **readFully**(byte [])
- void **readFully**(byte[] b, int off, int len)
- int **readInt**()
- long **readLong**()
- short **readShort**()

- int **readUnsignedByte**()
- int **readUnsignedShort**()
- String **readUTF**()

Con esta clase se ha podido probar el almacenamiento de todos los tipos primitivos de Java. En todos los métodos indicados anteriormente, excepto readFully() se ha de almacenar en el fichero de trazas el resultado del método. En el caso de readFully se ha de almacenar el valor del primer parámetro (es el mismo caso del método read()).

5.5.3.2. Reejecución

En el caso de la reejecución se siguió el mismo patrón de desarrollo, primero instrumentando los métodos de las clases básicas:

- **InputStream**
- **Reader**

En este caso, como lo que leemos del fichero son Strings y se ha de transformar en el tipo de dato que requiere cada uno de los métodos se necesitó una forma de diferenciar cada uno de los métodos (sobre todo los que tienen el mismo número de parámetros) para hacer la conversión del dato leído de forma correcta.

Para solucionar este problema se intentó obtener el tipo de dato que es cada parámetro (en este caso con comprobar el primero de ellos, que es el único que cambia fue más que suficiente). Javassist ofrece una variable durante la instrumentación ($sig) que nos da los tipos de dato de los parámetros de entrada de un método. De esta forma se pudieron separar las instrumentaciones. Sin embargo falta por comprobar si el formato del tipo de dato que devuelve esta variable es un formato estándar y se puede usar independientemente de la versión de la jvm.

Una vez distinguidos todos los métodos a instrumentar se vio que era necesario poder actualizar el valor de los parámetros que se pasan al método que está siendo instrumentado, ya que es ahí donde se devuelve la lectura. Por tanto los datos leídos del fichero de trazas debían ponerse en el primer parámetro de entrada de los métodos que funcionan así. Por suerte, javassist permite acceder a los parámetros de entrada de los métodos que están siendo instrumentados ($1,$2,....) y además permite que sean actualizados. Así fue sencillo instrumentar los métodos para obtener las lecturas del fichero de trazas y no del fichero físico.

Una vez solucionados los dos temas básicos se continuó con la instrumentación de las siguientes clases:

- BufferedReader
- DataInputStream

El caso de la clase BufferedReader tenía el problema de que el final del fichero en el caso de usar el método readLine() no viene indicado por un -1, sino que viene indicado por un null. Ya en la ejecución se había almacenado este valor como la cadena "null". Aunque ya se había comentado que esta solución no es definitiva (en el fichero podría existir esta cadena), se hizo la instrumentación suponiendo que al encontrar esta cadena se entendería como un final de fichero.

En el caso de la clase DataInputStream el problema es también la marca de fin de fichero. En este caso concreto el fin de fichero viene dado por el lanzamiento de una excepción (EndOfFileException). Por eso en este caso es más complicado saber cuando realmente finaliza el fichero. Este problema está aun por resolver. Aunque se había supuesto inicialmente que los procesos se ejecutarían sin excepciones, en este caso es inevitable ya que es la única forma de saber cuando se ha llegado al final del fichero de forma controlada.

A parte de instrumentar los métodos de lectura de datos, es necesario inhibir los métodos que obtienen datos del fichero o realizan acciones sobre ellos, ya que en la reejecución no se va a disponer de los ficheros físicos. Si se comprueba la existencia del fichero, etc...habrá que almacenarlo o indicarlo de alguna manera para que el flujo del programa sea como el original de la ejecución. Hay un gran número de métodos que se basan en la existencia del fichero y que se deberían inhibir.

Un problema importante que se encontró en este caso es que al instrumentar las llamadas a los métodos de lectura de datos, se instrumentaron involuntariamente los métodos de lectura del logger creado para almacenar los datos en el fichero de trazas. Al hacer esto e intentar la reejecución, se entraba en un bucle infinito. Para resolver este problema se necesitaba evitar la instrumentación de estos métodos. Una idea fue no instrumentar para la clase que usa el logger (RandomAccessFile), sin embargo haciendo esto no se instrumentarían tampoco en el caso de usar esta clase en la aplicación. La solución que se tomó finalmente fue la de evitar que las clases que pertenecían al paquete del logger fuesen instrumentadas. Con esto se solución el problema.

5.5.4. Todos los casos juntos

Hasta ahora se han desarrollado clases para transformar por separado los tres casos explicados anteriormente. Pero para afrontar la transformación de aplicaciones java reales se necesitaba un transformador que tuviese en cuenta todos los casos.

5.6. Desarrollo de las aplicaciones

Para el desarrollo de las aplicaciones se diseñó el siguiente modelo, en el que se muestran las clases básicas de todos los casos:

Las clases del modelo son las siguientes:

- BaseTranslator

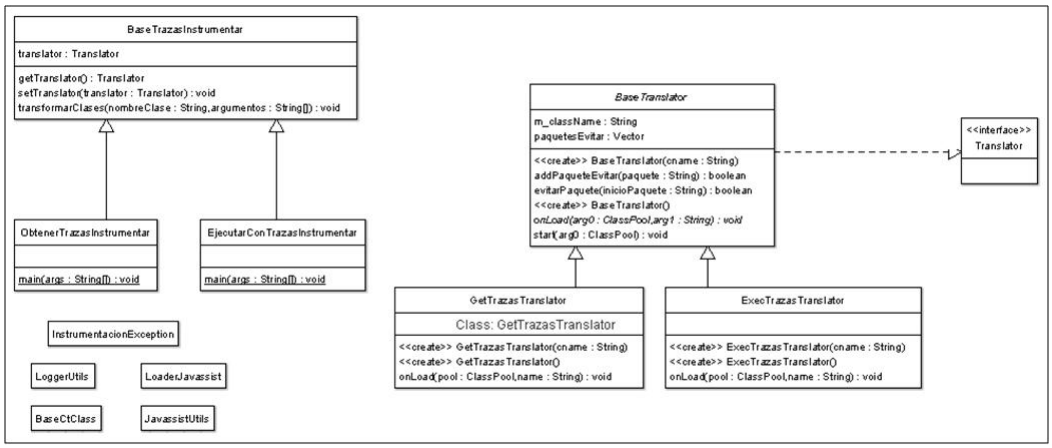


Figura 5.4: Diagrama básico de clases

- GetTrazasTranslator
- ExecTrazasTranslator
- BaseTrazasInstrumentar
- ObtenerTrazasInstrumentar
- EjecutarConTrazasInstrumentar
- Clases de soporte:
  - InstrumentationException
  - LoggerUtils
  - LoaderJavassist
  - BaseCtClass
  - JavassistUtils

BaseTrazasInstrumentar Es la base de la instrumentación del resto de clases.

Contiene un método transformarClases(String nombreClase, String [] argumentos):void, que realiza la instrumentación de la clase. Para ello la clase es cargada con un loader realizado para la aplicación. Este loader se llama LoaderJavassist y extiende de la clase Loader que viene en el jar de javassist. El loader carga la clase que se le indica y llama al método main de la misma, con los argumentos que se le indican a la función por parámetro. Al loader se le pasa una clase que realiza la instrumentación de las clases que se van cargando. Esta clase, que ha de ser o heredar de la clase Traslator del jar de javassist, es la que indica que transformaciones se han de realizar en las clases que se van cargando.

Esta clase es la base para la construcción de las clases de instrumentación para los dos tipos de ejecución que tratamos:

- **ObtenerTrazasInstrumentar:** Ejecución u obtener trazas en el fichero
- **EjecutarConTrazasInstrumentar:** Reejecución o ejecución del proceso cogiendo los datos del fichero de trazas

Estas dos clases son muy parecidas. El objetivo de las dos es indicar el traductor (Translator) a usar en la instrumentación y realizar la llamada al método que llama al main de la clase principal para que esta se cargue y ejecute. Se han creado dos clases distintas para mantener la independencia en las ejecuciones, cualquiera de las dos podría cambiar sin afectar a la otra.

BaseTranslator Como se ha indicado antes, a la clase base de instrumentación es necesario indicarle un traductor que le indique que ha de instrumentar en cada caso (dependiendo de la prueba que se haga).

Esta clase extiende de la clase *Translator* que viene en el jar de javassist y es abstracta, tienen un método a implementar en las clases que descienden de ella. El método abstracto es el siguientes:

public abstract void onLoad(ClassPool arg0, String arg1) throws NotFoundException, CannotCompileException ;

Este método se llama al cargar una clase que se va a ejecutar y en su cuerpo se debe indicar que hacer con la clase que se carga. Tiene dos parámetros: arg0 que indica que ClassPool ha de usar el Translator y arg1 que es el nombre de la clase que se está cargando.

Esta clase también tiene un método para añadir paquetes a evitar. Se pueden indicar múltiples paquetes a evitar y la clase se encarga de que al instrumentar un proceso, los paquetes indicados no sean instrumentados.

De ella heredan las siguientes clases:

- **GetTrazasTranslator:** Es el traductor para la ejecución u obtención de trazas en el fichero.
- **ExecTrazasTranslator:** Es el traductor para la reejecución o ejecución del proceso cogiendo los datos del fichero de trazas.

Estas dos clases implementan el método onLoad, para instrumentar las clases necesarias para cada uno de los casos.

5.7. Pruebas y test del sistema

5.7.1. Introducción

Se han realizado diferentes pruebas para comprobar la validez de las soluciones encontradas. Debido a que el desarrollo se ha dividido en diferentes fases, una para cada una de los casos por separado más una final con todos los casos

juntos, las pruebas también se han realizado en fases, donde en cada fase se han hecho pruebas de los componentes de software desarrollados.

Para realizar las pruebas de cada fase se ha definido un procedimiento común a todas ellas y que asegura la validez de los desarrollos realizados. Las fases de pruebas son las siguientes:

- Fase I: en esta fase se comprobó la obtención del fichero de trazas con datos de entrada a la aplicación desde el main del proceso principal
- Fase II: en esta fase se comprobó la obtención del fichero de trazas con datos de entrada de la base de datos.
- Fase III:en esta fase se comprobó la obtención del fichero de trazas con datos de entrada de distintos tipos de ficheros de datos almacenados en el disco del sistema que ejecuta la aplicación.
- Fase IV: en esta fase se comprobó la obtención del fichero de trazas con datos de entrada de todos los tipos de acceso a datos.

En los anexos se detallan los siguientes componentes:

- Datos de prueba: se han desarrollado clases de prueba con diferentes tipos de acceso a datos, para comprobar las soluciones. También se han definido los contenidos de la base de datos y de los ficheros para poder comprobar los resultados de los procesos.
- Las pruebas realizadas en cada una de las fases.

5.7.2. Procedimiento

El procedimiento seguido para realizar las pruebas fue el mismo en cada caso (para cada problema: datos de entrada, acceso a bbdd, acceso a ficheros y todos los casos juntos). En la figura 5.5 se puede ver un esquema del procedimiento seguido.

El procedimiento es el siguiente:

1. Se genera una clase de prueba que realiza el acceso o funcionalidad que queremos probar (por ejemplo, accede a una base de datos y pinta los resultados, lee datos de un fichero, etc...). Se ejecuta el proceso de prueba para obtener el resultado. Este resultado se usará para comprobar en los siguientes pasos que la funcionalidad del proceso de prueba no cambia con la instrumentación. A este proceso de prueba le llamamos **proceso I**.
2. Se genera el proceso que realiza la instrumentación del proceso I para obtener un fichero que almacene las trazas de los datos de entrada a la aplicación. A este proceso le llamamos **proceso II**. Ejecutamos el proceso II y obtenemos los siguientes datos de la ejecución: el fichero con las trazas de los datos de entrada, el resultado del proceso y el conjunto de trazas de la ejecución del proceso. Comprobamos que el resultado del proceso

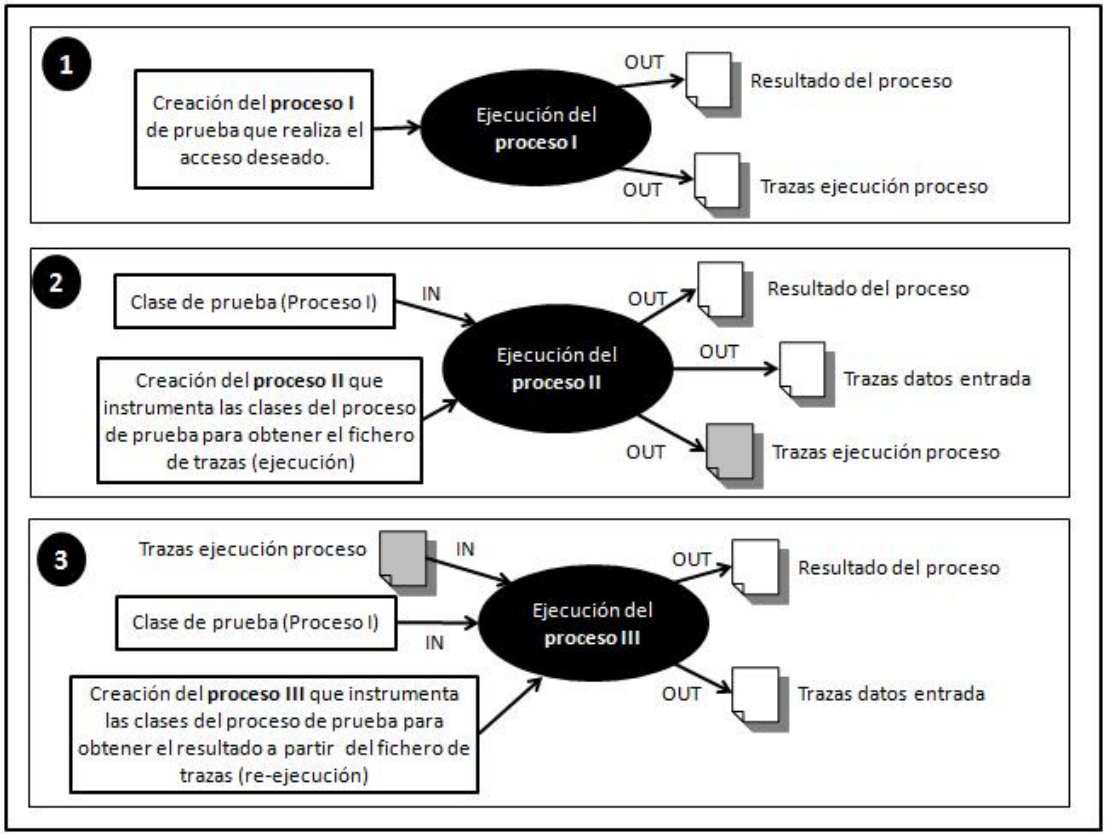


Figura 5.5: Proceso de pruebas

I es igual al del proceso II, para ver que la funcionalidad del proceso de prueba no ha sido modificado. Comprobamos las trazas almacenadas con los datos de entrada, para ver que se ha generado correctamente. Comprobamos el conjunto de trazas de la ejecución del proceso que nos muestran el conjunto de clases y métodos instrumentados, para comprobar que estos son correctos.

3. Se genera el proceso que realiza la instrumentación de la clase para que la ejecución no obtenga los datos de las fuentes originales, si no que las obtenga del fichero anteriormente generado con el proceso II. A este proceso le llamamos **proceso III**. Ejecutamos el proceso III, usando el fichero de trazas como datos de entrada. Obtenemos los siguientes datos de la ejecución: el resultado del proceso y el conjunto de trazas de la ejecución del proceso. Comprobamos que el resultado de los tres procesos es el mismo, para comprobar que la funcionalidad del proceso de prueba no ha cambiado con la instrumentación. Comprobamos el conjunto de trazas de la ejecución del proceso que nos indica el conjunto de clases y métodos que han sido instrumentados, para comprobar que estos son los correctos y que realmente los datos de entrada se están obteniendo del fichero y no de las fuentes originales. El fichero de trazas se puede modificar manualmente, para comprobar que con la ejecución del proceso III, la aplicación se comporta como si los datos fuesen solo los del fichero de trazas, lo que asegura que la instrumentación funciona de forma correcta.

En las pruebas los datos que se obtienen son los siguientes:

- **Resultado del proceso:** estos datos son dependientes del proceso que se prueba. Se trata del resultado de las operaciones definidas con los datos que entran a la aplicación, ya sea en el main del proceso principal, desde la base de datos o desde un fichero almacenado en disco. Estos resultados nos permitirán comprobar que el proceso que está siendo probado no ve modificada su funcionalidad al pasar por los diferentes procesos de instrumentación. Es necesario, para asegurar la validez de las soluciones, que los procesos originales no se vean modificados en funcionalidad, solamente se han de añadir las sentencias necesarias para la obtención de trazas.

- **Fichero de trazas de datos de entrada:** es el fichero con los datos de entrada a la aplicación. Este fichero se genera durante la ejecución del proceso, para almacenar los datos de entrada a la aplicación y se usa en la reejecución para comprobar que la ejecución ha sido correcta.

- **Trazas de ejecución del proceso:** estos datos se obtienen durante las instrumentaciones de los procesos de prueba. Nos indican las clases y métodos que van siendo modificados, los que son evitados porque no han de ser instrumentados y el conjunto de incidencias que pueden ocurrir durante la instrumentación (tanto para la ejecución como para la reejecución). Son útiles para comprobar que el conjunto de clases y métodos instrumentados son los correctos.

En los siguientes capítulos se detallan los datos, clases y métodos generados para llevar a cabo las pruebas y también los resultados obtenidos durante las mismas. Los tres tipos de datos definidos (resultado del proceso, fichero de trazas de datos de entrada y trazas de ejecución del proceso) nos ayudarán a validar las pruebas y decidir la validez del desarrollo realizado.

5.7.3. Conclusiones

El objetivo de las pruebas ha sido el de probar los desarrollos realizados para comprobar la validez de los resultados obtenidos. El objetivo inicial del desarrollo ha sido el de encontrar una forma de almacenar los datos de entrada/salida de una aplicación.

Con los datos de test y las clases de prueba (simples) desarrolladas, se ha comprobado que se instrumentan de forma correcta las clases para obtener los datos de entrada/salida definidos en un principio (datos de entrada a la aplicación, acceso a bases de datos y acceso a datos almacenados en ficheros). En algunos de los casos incluso se ha conseguido más de una solución posible al problema inicial. En estos casos, el uso de uno u otro método se basaría únicamente en el performance conseguido por la aplicación en cada caso y en la forma en la que se lanzaría un proceso cualquiera (ya que en algunos casos se requiere de una clase intermedia que realice posteriormente la llamada a la aplicación que realmente queremos ejecutar).

Todo esto se ha probado bajo el supuesto de que las aplicaciones se ejecutan en un entorno controlado y sin errores (los cambios de flujo que producen estos en una aplicación no se han tenido en cuenta en esta fase de desarrollo, ya que no son datos de entrada/salida en si, pero si es necesario almacenar una marca de donde se cambia el flujo de la aplicación a causa de una excepción cualquiera). Estas suposiciones limitan bastante la aplicación de las soluciones encontradas, pero es una base sobre la que seguir desarrollando para encontrar una solución aplicable a aplicaciones reales.

En definitiva podemos decir que el objetivo del desarrollo (el almacenar los datos de entrada/salida) de una aplicación se ha conseguido de forma satisfactoria. Sin embargo, el conjunto de suposiciones fijadas (versión jre, ejecución sin errores) hace que la solución sea, aún, no aplicable en aplicaciones reales.

Capítulo 6

Conclusiones y líneas futuras

6.1. Conclusiones

El objetivo del desarrollo ha sido encontrar y probar una forma de almacenar los datos de entrada de una aplicación cualquiera para poder reejecutarla a partir, únicamente, del fichero de trazas obteniendo el mismo resultado. De esta forma sería posible detectar algún tipo de ataque de un host a un agente que se ejecuta en su dominio. Ha quedado visto que para los accesos a datos más frecuentes es posible obtener los datos de entrada de forma sencilla. Sin embargo, para conseguir los objetivos propuestos, este desarrollo se ha realizado fijando un conjunto de variables a eliminar en posteriores fases de desarrollo. Estas variables han sido la versión de la jre usada en la ejecución/reejecución, el seguro de ejecutar las aplicaciones siempre sin errores (ni excepciones tanto controladas como incontroladas) y la instrumentación on-line en el momento de la ejecución de las aplicaciones.

La versión de la jre con la que se ejecuten las aplicaciones es importante, debido a que en cada nueva versión se introducen nuevas clases de acceso a datos (como puede ser el package java.nio que es nuevo en la versión 6.0 de la jre y que permite el acceso a datos de una forma distinta a como se ha hecho hasta la versión 1.5 de la misma). En cualquier caso, se debería usar la misma versión en la ejecución como en la reejecución. Solo de esta forma se aseguraría el correcto funcionamiento de la reejecución. En el caso de nuevas clases de acceso a datos de entrada y salida, se deberían analizar para obtener los métodos a instrumentar en cada caso y añadir al código de instrumentación las sentencias necesarias.

Sobre la ejecución de aplicaciones sin errores, el hecho de lanzarse un error en la aplicación implica una modificación del flujo de ejecución que debería poder reproducirse también en la reejecución de aplicaciones, durante la cual se intenta verificar que la aplicación ha sido ejecutada de forma correcta. Estas excepciones se deberían apuntar en el fichero de trazas para indicar el cambio en el flujo, sin embargo no se han tenido en cuenta en esta fase de desarrollo.

Para la instrumentación de aplicaciones reales se debería tener en cuenta la gestión de errores. En caso contrario, la solución no podría darse por buena, ya que solo funcionaría en aplicaciones ejecutadas sin errores. El tener errores es algo inevitable en aplicaciones, más aún cuando son las aplicaciones las que viajan de host en host, ejecutándose en diferentes entornos que pueden no estar correctamente configurados o tener falta de recursos necesarios para la correcta ejecución de un agente.

El momento seleccionado para la instrumentación de las clases se ha seleccionado teniendo en cuenta que el objetivo del proyecto ha sido la de encontrar una forma de obtener las trazas. Por esto, de las dos formas de instrumentación de una aplicación, "en tiempo de construcción" (instrumentando las clases y guardando estas modificadas añadiendo las sentencias de traceado) y "en tiempo de ejecución" (instrumentando las clases en el momento de cargarlas en la ejecución) se ha seleccionado la segunda por la facilidad que implica el testeo de las soluciones, al no tener que almacenar clases modificadas y asegurarse de que se llama a estas en lugar de las originales. Esta selección implica que en el momento de la ejecución se llama a una clase intermedia pasándole el nombre de la clase a ejecutar, en lugar de llamar a la aplicación por medio del main de su clase principal. Esto no es problema, ya que si se decidiese implementar este método de traceado, el método de instrumentación a seleccionar sería el de "en tiempo de construcción". Con este método se podría crear una aplicación de instrumentación que leyese e instrumentase las clases necesarias, sin necesidad de instrumentarlas en cada ejecución. Esta aplicación guardaría los jars con las nuevas clases instrumentadas y serian estas las que se distribuirían, de forma que el agente a ejecutar ya llevaría incorporadas las sentencias de traceado. Esta solución evitaría también el tener que llamar a una aplicación a través de una clase intermedia (que es la que hace la instrumentación de lo que se ejecuta).

Tampoco se han tenido en cuenta consideraciones de rendimiento de las aplicaciones, aunque se supone que la adición de un conjunto de sentencias para el traceado de datos de entrada y salida no debería perjudicar el performance de ninguna aplicación en exceso (inclusive en aplicaciones que manejan muchos datos de entrada y salida). Otros temas que no se han tenido en cuenta son, por ejemplo, el tamaño que puede alcanzar un fichero o la forma de distinguir ficheros entre distintas ejecuciones. En el caso del tamaño hemos supuesto que el host que ejecuta el agente tiene los suficientes recursos como para que no falle la ejecución por falta de espacio para el fichero de trazas. Lo que si puede ser un problema es el lugar de almacenamiento en este host ejecutor, ya que es posible que no haya permisos para guardarlo y no podríamos comprobar la correcta ejecución de un agente en este host. Respecto a la forma en la que se deberían distinguir dos ejecuciones distintas, es complicado de resolver. En un principio parece sencillo debido a que a cada agente se le asigna un identificador único. Sin embargo un agente puede ejecutarse n veces en un host y no sabríamos distinguir cual es la ejecución que queremos obtener.

Las consideraciones anteriores valen para cualquier tipo de aplicación (o agente que no se comunique con otros agentes del sistema). En el caso de agentes se debería tener en cuenta la comunicación entre ellos, guardando los datos

que vienen de fuera (respuestas de otros agentes) para que la reejecución fuese correcta.

Con todas estas consideraciones, que hemos tenido en cuenta para el desarrollo, vemos que falta solucionar muchos temas para que la solución sea viable para aplicaciones reales ejecutándose en diferentes tipos de entorno.

6.2. Líneas futuras

Como líneas futuras se podrían tener en cuenta los siguientes temas sin resolver:

- Generación de una aplicación de instrumentación
- Gestión de errores de las aplicaciones
- Comunicación entre agentes
- Gestión y tratamiento de ficheros de trazas

Generación de una aplicación de instrumentación Como se ha comentado en los capítulos de desarrollo y conclusiones, se han realizado las pruebas instrumentando las clases a ejecutar "en tiempo de ejecución". Esto es, se ejecuta la aplicación y una aplicación intermedia va instrumentando las clases de la aplicación tal y como van siendo cargadas por la JVM (Java Virtual Machine). Este método es poco flexible por dos motivos: se necesita una clase intermedia que llame a la que realmente se quiere ejecutar y la pérdida de tiempo que conlleva la instrumentación, así como los errores que esta comporta.

Para resolver este problema se debería instrumentar las clases de la aplicación a ejecutar antes de la ejecución ("en tiempo de ejecución") de forma que antes de la ejecución ya se introducen las sentencias de traceo necesarias. Para realizar esta instrumentación se podría crear una aplicación que cogiese las clases de la aplicación (o el jar / jars que la contengan) y devuelva el jar o jars con las clases ya instrumentadas a punto para ser ejecutadas. Estas clases modificadas son las que se distribuirían en lugar de las originales. En estas nuevas, la funcionalidad no habría cambiado, únicamente se habrían introducido las sentencias para recoger los valores de los datos de entrada/salida.

Gestión de errores de las aplicaciones Como se ha comentado en los capítulos anteriores, si se lanza una excepción durante la ejecución de una aplicación se modifica el flujo de ejecución de la misma. Esto significa que las sentencias ejecutadas de la aplicación no son las mismas que las de la aplicación ejecutada sin errores. Para poder reejecutar la aplicación de la misma forma que se ha ejecutado se deberían guardar marcas de las excepciones que se han lanzado.

Estas excepciones no son realmente datos de entrada/salida de la aplicación, por lo tanto no deberían ir en el fichero de trazas. Sin embargo, a priori, no tenemos otro sitio donde indicarlas. El lugar donde ponerlas sería ahí, ya que

los resultados guardados en el fichero también dependen del flujo de ejecución de la aplicación y de las sentencias que se hayan ejecutado durante la ejecución.

Comunicación entre agentes No se han tenido en cuenta en este desarrollo las características propias de los agentes (basados en aglets) como podría ser la comunicación entre ellos.

Para una correcta reejecución de un agente se deberían tener almacenados los datos que vienen de otros agentes si se ha intentado la comunicación con otros agentes del sistema. Se debería analizar con detalle las opciones de comunicación de un agente para añadir la instrumentación adecuada.

Gestión y tratamiento de ficheros de trazas En el protocolo definido en 7.10 se define como enviar los ficheros de trazas de forma segura (sin que sufra modificaciones), por medio de funciones Hash y cifrado de los contenidos con las claves privadas y públicas de los elementos que participan en la comunicación (el propietario del agente y el host ejecutor). Sin embargo, no se especifica una metodología para nombrar los ficheros de ejecución, el almacenamiento en el host ejecutor y la forma de pedir (el emisor del agente pedirá el fichero de trazas en caso de dudar del host ejecutor) y enviar (el host ejecutor enviará el fichero correspondiente a la ejecución pedida) los ficheros.

Esta metodología es necesaria para aplicar la solución a aplicaciones reales. En un momento dado no sabemos cuantos agentes pueden ser ejecutados por el host ejecutor o incluso si alguno se ejecuta más de una vez. Es necesario saber como se llama cada fichero, donde almacenarlo, el tiempo que es necesario mantener en el host ejecutor por si se pide, etc...para poder llevar a cabo una reejecución en caso de duda.

Capítulo 7

Anexo 1: Pruebas

7.1. Introducción

En este capítulo se detallan las pruebas realizadas para validar los componentes de software desarrollados. El procedimiento seguido en todos los casos, se detalla en el capítulo 5.7.2.

En los siguientes capítulos se detallan los siguientes componentes:

- Datos de prueba: se han desarrollado clases de prueba con diferentes tipos de acceso a datos, para comprobar las soluciones. También se han definido los contenidos de la base de datos y de los ficheros para poder comprobar los resultados de los procesos.
- Las pruebas realizadas en cada una de las fases.

7.2. Configuración y datos

Para las pruebas se han usado los siguientes datos:

- Creación de una base de datos con un conjunto de tablas.
- Creación de diferentes ficheros.

A continuación se describen las estructuras creadas y los datos que se han cargado.

7.2.1. Base de datos

Para las pruebas se ha creado dos tablas:

- personas (7.1). Cargada con los datos de la 7.2.
- extracto (7.3). Cargada con los datos de la 7.4.

Las dos tablas están relacionadas entre si por el NIF.

Cuadro 7.1: Tabla personas

Campo	Tipo	Null	Defecto
Nombre	varchar(20)	Yes	NULL
Apellido	varchar(40)	Yes	NULL
Edad	int(3)	Yes	NULL
Sexo	char(1)	Yes	NULL
Fecha_nacimiento	date	Yes	NULL
NIF	varchar(9)	No	NULL

Cuadro 7.2: Datos de la tabla personas

Nombre	Apellidos	Edad	Sexo	Fecha_nacimiento	NIF
Eva	Soto	30	M	2008-12-03	07721443M
Maria	Gonzalez	15	M	1983-04-25	12345678J
Pedro	Sánchez	32	H	1976-11-03	12354678J

Cuadro 7.3: Tabla extracto

Campo	Tipo	Null	Defecto
NIF	varchar(9)	No	NULL
Disponible	FLOAT	Yes	NULL
Actualizado	BOOLEAN	Yes	NULL

Cuadro 7.4: Datos de la tabla extracto

NIF	Disponible	Actualizado
07721443M	111.111	1
12345678J	8571.43	0
12354678J	44166.7	1

7.2.2. Ficheros

Para las pruebas de acceso a ficheros se usaron diferentes ficheros de pruebas en función del tipo de dato a probar.

Para probar las clases FileInputStream, BufferedReader y FileReader fue suficiente con un fichero de texto normal con un texto.

En el caso de la prueba de DataInputStream, el fichero se tuvo que crear con la clase DataOutputStream. De esta forma los valores se escriben el fichero con los métodos que los almacenan como tipos primitivos de Java. Si no se genera el fichero de esta forma, luego no puede leerse el fichero como tipos primitivos de Java de forma correcta.

7.2.3. Proceso de prueba para aplicaciones completas

Para probar la instrumentación de una aplicación real, teniendo en cuenta las tres posibilidades:

- Entrada de datos en el main de la aplicación
- Acceso a datos en ficheros
- Acceso a datos en bases de datos

Se creó la estructura de datos mostrada en 7.1.

La aplicación creada funciona de la siguiente manera:

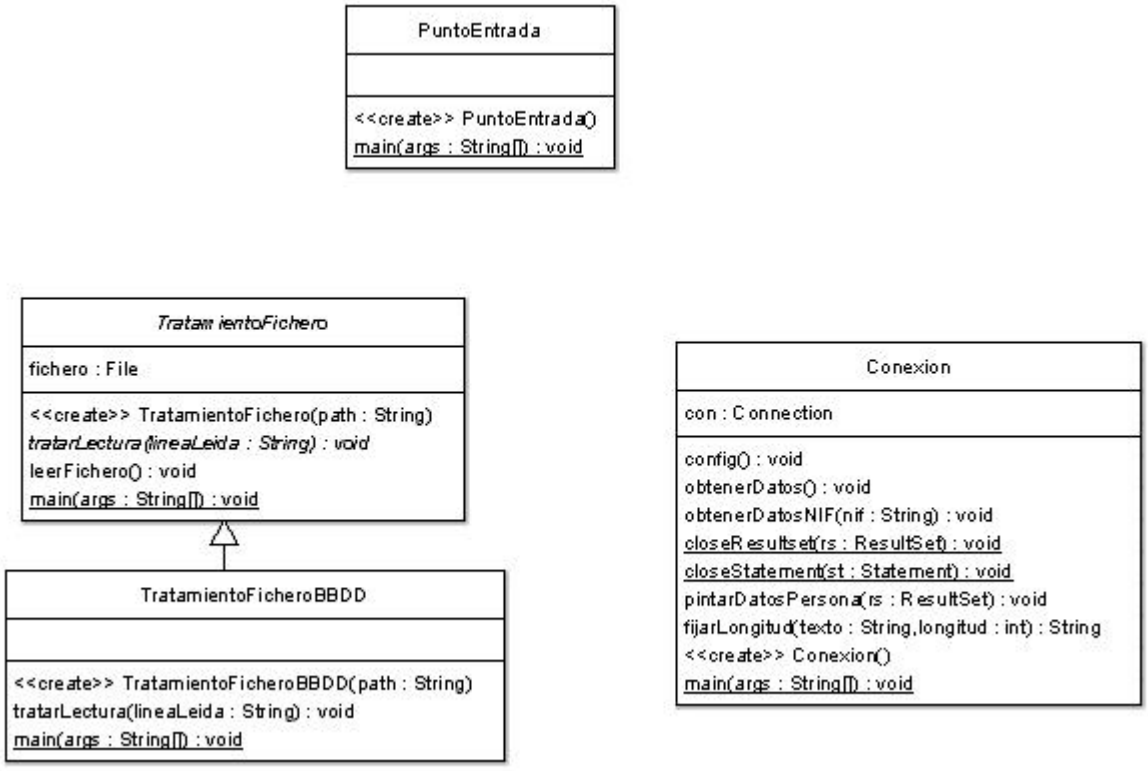
- La clase PuntoEntrada.java recoge los valores de entrada de la aplicación en su método main, crea la clase TratamientoFicheroBBDD y llama a su método leerFichero.
- La clase TratamientoFicheroBBDD extiende de la clase TratamientoFichero, que es abstracta y tiene un método llamado leerFichero. Este método, leerFichero, lee el fichero que se le pasa por parámetro al método y para cada línea leída llama al método tratarLectura.
- El método tratarLectura es abstracto en la clase TratamientoFichero. En la clase TratamientoFicheroBBDD se ha implementado para que por cada línea leída (que se supone que es un NIF de persona) busque en la tabla los datos de la persona la que pertenece este NIF.
- La búsqueda de datos se realiza a través de la clase Conexión.

Con esto tenemos una aplicación de cuatro clases que realizan los 3 tipos de accesos.

7.3. Datos de entrada de la aplicación

Para obtener los datos de entrada de la aplicación se desarrolló el código necesario tanto con la API Reflection de Java como con Javassist, por tanto se hicieron pruebas con los dos desarrollos.

Figura 7.1: Modelo de datos para probar al instrumentación de aplicaciones reales (con todos los casos)



Para probar el desarrollo se usó una clase que realiza una suma con un conjunto de números que se le pasa por parámetro. Como hemos comentado, los parámetros de entrada de la aplicación son siempre de tipo String y es competencia del programador adecuar el valor al tipo necesario, por tanto es la aplicación la que hace la conversión. Para la solución solo tenemos que tener en cuenta datos de tipo String.

La clase de prueba es *es.pfc.trazas.clasessoporte.SumaArgumentos*. Convierte a enteros los valores que se pasan por parámetro y los suma, pintando la operación y el resultado de la misma por pantalla.

Para los parámetros de entrada:

51 462 516 174 1035 900 1110 1379 557 495 1346 654

El resultado de la ejecución es el siguiente:

Suma: 51 + 462 + 516 + 174 + 1035 + 900 + 1110 + 1379 + 557 + 495 + 1346 + 654
Resultado: 8679

7.3.1. API Reflection

Para la ejecución se usó la clase:

es.pfc.trazas.reflection.main.ObtenerTrazas

Esta clase obtiene los datos de entrada de la clase mediante la API reflection de java y los almacena en el fichero indicado en el fichero de propiedades.

Para los parámetros de entrada:

es.pfc.trazas.clasessoporte.SumaArgumentos 51 462 516 174 1035 900 1110 1379 557 495 1346 654

El resultado de la ejecución es el siguiente:

Clase "es.pfc.trazas.clasessoporte.SumaArgumentos"
Fichero: D:|Software|PFC|workspace|PFC|trazas.dat
El fichero existe, lo borramos...
Suma: 51 + 462 + 516 + 174 + 1035 + 900 + 1110 + 1379 + 557 + 495 + 1346 + 654
Resultado: 8679

Se indica en cada línea:

1. Clase que se ha de ejecutar.
2. Path completo del fichero de trazas que se genera.
3. Indica que el fichero ya existía de otra ejecución anterior, se elimina y se crea un fichero nuevo vacío que contendrá los datos únicamente de la ejecución actual.
4. La operación que se va a realizar.
5. Resultado de la operación.

En el fichero de trazas quedaron registrados los valores de entrada de la aplicación (todas como índice 0 porque pertenecen a un array):

0 51
0 462
0 516

0 174
0 1035
0 900
0 1110
0 1379
0 557
0 495
0 1346
0 654

Para la reejecución se usó la clase:

es.pfc.trazas.reflection.main.EjecutarConTrazas

Esta clase obtiene los datos de entrada de la clase mediante la API reflection de java y los almacena en el fichero indicado en el fichero de propiedades.

Para los parámetros de entrada:

es.pfc.trazas.clasessoporte.SumaArgumentos

El resultado de la ejecución es el siguiente:

Clase "es.pfc.trazas.clasessoporte.SumaArgumentos"
Fichero: D:\Software\PFC\workspace\PFC\trazas.dat
Suma: 51 + 462 + 516 + 174 + 1035 + 900 + 1110 + 1379 + 557 + 495 + 1346 + 654
Resultado: 8679

Se indica en cada línea:

1. Clase que se ha de ejecutar.
2. Path completo del fichero de trazas que se utiliza para leer los parámetros a usar.
3. La operación que se va a realizar.
4. Resultado de la operación.

El resultado de la reejecución ha sido el mismo que el de ejecutar la aplicación sin usar Reflection.

7.3.2. Javassist

7.3.2.1. Ejecución

Aplicación de instrumentación Para instrumentar el main de la aplicación principal y obtener trazas de los parámetros de entrada, se desarrolló la siguiente clase:

es.pfc.trazas.javassist.exec.main.ObtenerTrazas

Esta clase carga la clase que se le pasa por parámetro, almacena los valores de los parámetros de entrada y ejecuta su main. Estos datos se almacenan en el fichero de trazas, de forma que pueden ser recuperados posteriormente para la reejecución.

Aplicación de prueba Para la ejecución se desarrolló una aplicación de una única clase que obtiene los datos de entrada, los parsea (ya que espera números), los suma y pinta el resultado que obtiene por pantalla. Esta aplicación se pasa (el nombre completo) a la aplicación que instrumenta en los parámetros de entrada del main de la clase principal. La clase es la siguiente:

es.soporte.trazas.ejecutables.SumaArgumentos (la clase de prueba)

A esta clase le pasamos los siguientes parámetros:

51 462 516 174 1035 900 1110 1379 557 495 1346 654

Resultado de la ejecución El resultado de la ejecución es el siguiente:

Suma: 51 + 462 + 516 + 174 + 1035 + 900 + 1110 + 1379 + 557 + 495 + 1346 + 654

Resultado: 8679

Se puede comprobar que el resultado es el mismo que en la ejecución de la clase de forma independiente. Por tanto, la funcionalidad de la clase no ha cambiado. Esto es así porque en la instrumentación de la clase no modificamos el comportamiento, solo añadimos más (el almacenamiento de los datos en el fichero de trazas).

Trazas de la ejecución Las trazas de ejecución de la aplicación son las siguientes. En ellas se indican los métodos que se instrumentan:

Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.SumaArgumentos"

Instrumentamos método "main" de la clase ""
Fichero: D:|Software|PFC|workspace|PFC|trazas.dat
El fichero existe, lo borramos...

Se indica en cada línea:

1. En esta línea se indica la clase que se instrumenta.
2. En esta línea se indica el método que se instrumenta (como no se dice nada, sobreentendemos que es el método main el que se instrumenta).
3. Path completo del fichero de trazas que se genera.
4. Indica que el fichero ya existía de otra ejecución anterior, se elimina y se crea un fichero nuevo vacío que contendrá los datos únicamente de la ejecución actual.

Contenido del fichero de trazas El contenido del fichero de trazas es el que se muestra a continuación:

0 51
0 462
0 516
0 174
0 1035
0 900

0 1110
0 1379
0 557
0 495
0 1346
0 654

En el fichero de trazas quedaron registrados los valores de entrada de la aplicación (todas como índice 0 porque pertenecen a un array)

7.3.2.2. Reejecución

Aplicación de instrumentación Para instrumentar las clases de acceso a bbdd se desarrolló la siguiente clase:

es.pfc.trazas.javassist.main.EjecutarConTrazas

Esta clase carga la clase que se le pasa por parámetro, obtiene los datos del fichero de trazas y ejecuta su main con estos datos. En este caso los métodos se instrumentan para obtener los datos del fichero de trazas en lugar de obtenerlos de la base de datos, de forma que se puede comprobar si la ejecución ha sido correcta.

Aplicación de prueba Para la ejecución se desarrolló una aplicación de una única clase que obtiene los datos de entrada, los parsea como enteros (ya que espera números), los suma y pinta el resultado que obtiene por pantalla. Esta aplicación se pasa (el nombre completo) a la aplicación que instrumenta en los parámetros de entrada del main de la clase principal. La clase es la siguiente:

es.soporte.trazas.ejecutables.SumaArgumentos(la clase de prueba)

La clase es la misma que en la ejecución, ya que intentamos comprobar la reejecución con los datos del fichero.

Resultado de la ejecución El resultado de la ejecución es el siguiente:

Suma: 51 + 462 + 516 + 174 + 1035 + 900 + 1110 + 1379 + 557 + 495 + 1346 + 654

Resultado: 8679

Se puede comprobar que el resultado es el mismo que en la ejecución de la clase de forma independiente. Por tanto, la funcionalidad de la clase no ha cambiado. Esto es así porque en la instrumentación de la clase no modificamos el comportamiento, solo añadimos más (el almacenamiento de los datos en el fichero de trazas).

Trazas de la ejecución Las trazas de ejecución de la aplicación son las siguientes. En ellas se indican los métodos que se instrumentan:

Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.SumaArgumentos"

Instrumentamos método "main" de la clase ""
Fichero: D:\Software\PFC\workspace\PFC\trazas.dat

El total de variables leídas: 0
Se indica en cada línea:

1. Clase que se instrumenta.
2. Método de la clase que se instrumenta. Como no se indica nombre sobreentendemos que se instrumenta el método main de la aplicación.
3. Path completo del fichero de trazas que se utiliza para leer los parámetros a usar.
4. El número de variables leídas.

Las trazas de ejecución indican que los métodos se han instrumentado correctamente.

7.4. Acceso a base de datos

Para probar el acceso a base de datos se desarrolló una clase simple que realiza los pasos básicos:

1. Establecimiento de la conexión
2. Lanzamiento de la query
3. Obtención de datos y tratamiento. En este paso, se pintan por pantalla los datos de cada persona: nombre, edad, NIF, disponible, actualizado.

La clase desarrollada se llama *es.pfc.trazas.clasessoporte.Conexión* y tiene un método main que realiza la conexión y obtiene datos de la tabla de personas creada.

El resultado de la ejecución de esta aplicación es el siguiente:

————————————————————————————————

Nombre: Eva | Edad: 30 | NIF: 07721443M | Disponible: 111.111 | Actualizado: true

Nombre: Maria | Edad: 15 | NIF: 12345678J | Disponible: 8571.43 | Actualizado: false

Nombre: Pedro | Edad: 32 | NIF: 12354678J | Disponible: 44166.7 | Actualizado: true

————————————————————————————————

Con esta clase obtenemos los siguientes tipos de datos:

- String (nombre y NIF)
- int (Edad)
- float (Disponible)
- boolean (Actualizado)

7.4.1. Ejecución

Aplicación de instrumentación Para instrumentar las clases de acceso a bbdd se desarrolló la siguiente clase:

es.pfc.trazas.javassist.exec.bbdd.ObtenerTrazasInstrumentar

Esta clase carga la clase que se le pasa por parámetro, ejecuta su main e instrumenta las llamadas a los métodos de consulta de datos a la base de datos. Estos datos se almacenan en el fichero de trazas, de forma que pueden ser recuperados posteriormente para la reejecución.

Aplicación de prueba Para la ejecución se desarrolló una aplicación de una única clase que realiza el acceso a la base de datos y pinta los datos que obtiene por pantalla. Esta aplicación se pasa (el nombre completo) a la aplicación que instrumenta en los parámetros de entrada del main de la clase principal. La clase es la siguiente:

es.soporte.trazas.ejecutables.bbdd.Conexion (la clase de prueba)

Resultado de la ejecución El resultado de la ejecución es el siguiente:

————————————————————————————————————

Nombre: Eva | Edad: 30 | NIF: 07721443M | Disponible: 111.111 | Actualizado: true

Nombre: Maria | Edad: 15 | NIF: 12345678J | Disponible: 8571.43 | Actualizado: false

Nombre: Pedro | Edad: 32 | NIF: 12354678J | Disponible: 44166.7 | Actualizado: true

————————————————————————————————————

Se puede comprobar que el resultado es el mismo que en la ejecución de la clase de forma independiente. Por tanto, la funcionalidad de la clase no ha cambiado. Esto es así porque en la instrumentación de la clase no modificamos el comportamiento, solo añadimos más (el almacenamiento de los datos en el fichero de trazas).

Trazas de la ejecución Las trazas de ejecución de la aplicación son las siguientes. En ellas se indican los métodos que se instrumentan:

Evita paquete:com.mysql.jdbc

Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.bbdd.Conexion"

Instrumentado llamada al método: "getString" de la clase: "java.sql.ResultSet"

Instrumentado llamada al método: "getInt" de la clase: "java.sql.ResultSet"

Instrumentado llamada al método: "getString" de la clase: "java.sql.ResultSet"

Instrumentado llamada al método: "getFloat" de la clase: "java.sql.ResultSet"

Instrumentado llamada al método: "getBoolean" de la clase: "java.sql.ResultSet"

Instrumentado llamada al método: "next" de la clase: "java.sql.ResultSet"
Instrumenta llamada a métodos de la clase: "es.pfc.log.Logger"
Instrumenta llamada a métodos de la clase: "es.pfc.log.exception.LoggerException"

Fichero: D:|Software|PFC|workspace|PFC|trazas.dat
El fichero existe, lo borramos...
Se indica en cada línea:

- 1 - En esta línea se indica un paquete a evitar. En este caso evita el jar del fabricante de la base de datos, donde se hace el acceso real, ya que si se instrumentan estas clases se dejaría de hacer el acceso a las tablas reales.
- 2 - En esta línea se indica que clase se instrumenta (*es.soporte.trazas.ejecutables.bbdd.Conexion*), que en nuestro caso es la clase de prueba.
- 3 a 8 - En estas líneas se indican que llamadas a métodos de la clase de prueba se instrumenta.
- 9 a 10 - En estas líneas se indican otras clases en las que se buscan llamadas a métodos de acceso a bbdd. En este caso son las clases del Logger, que se cargan para almacenar los datos en el fichero de trazas. Estas clases no se evitan porque en el caso de base de datos no hay acceso por parte de ellas a las tablas. Si no fuese así deberían estar en la lista de paquetes a evitar. En las trazas se comprueba que no se han encontrado llamadas a métodos de bbdd ya que no hay trazas de instrumentación de ninguna llamada a ningún método.
- 11 - Path completo del fichero de trazas que se genera.
- 12 - Indica que el fichero ya existía de otra ejecución anterior, se elimina y se crea un fichero nuevo vacío que contendrá los datos únicamente de la ejecución actual.

En la imagen 7.2 se puede ver que los métodos instrumentados son los correctos. Según las trazas se hacen las siguientes llamadas a los siguientes métodos:

- next()
- getString()
- getInt()
- getString()
- getFloat()
- getBoolean()

Estas llamadas son las que dicen las trazas que se instrumentan, por lo tanto la instrumentación parece correcta. Para comprobar que realmente se han instrumentado bien los métodos hay que mirar que en el fichero de trazas se hayan almacenado los valores de las variables que se leen.

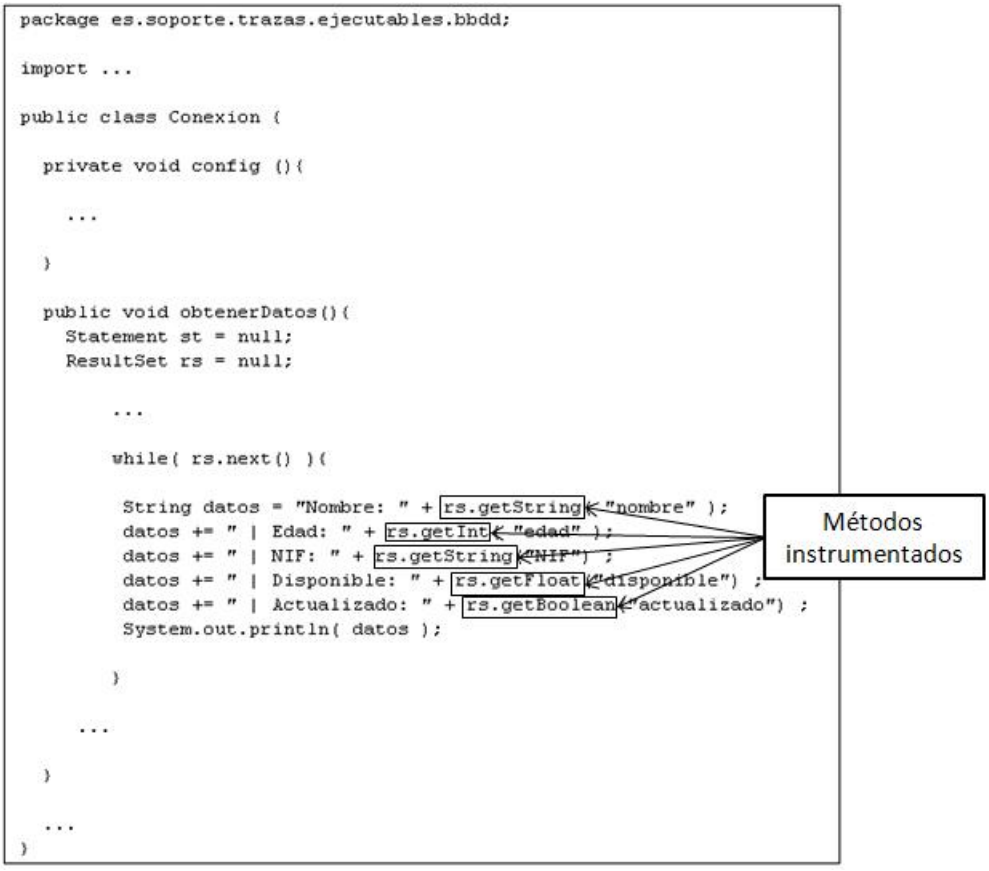


Figura 7.2: Clase de prueba para acceso a base de datos

Contenido del fichero de trazas El contenido del fichero de trazas es el siguiente:

0 true
1 Eva
2 30
3 07721443M
4 111.111
5 true
6 true
7 Maria
8 15
9 12345678J
10 8571.43
11 false
12 true
13 Pedro
14 32
15 12354678J
16 44166.7
17 true
18 false

Los valores 0, 6, 12 y 18 indican si existe resultado (es el next() del objeto ResultSet). Los valores 1, 7 y 13 corresponden a los nombres de las personas, los 2, 8 y 14 corresponden a las edades, los 3, 9 y 15 corresponden a los NIF, los 4, 10 y 16 corresponden a disponible y los 5, 11 y 17 corresponden a actualizado. Son los datos que se han obtenido con la clase de prueba.

7.4.2. Reejecución

7.4.2.1. Reejecución con el fichero generado en la ejecución

Aplicación de instrumentación Para instrumentar las clases de acceso a bbdd se desarrolló la siguiente clase:

es.pfc.trazas.javassist.exec.bbdd.EjecutarConTrazasInstrumentar

Esta clase carga la clase que se le pasa por parámetro, ejecuta su main e instrumenta las llamadas a los métodos de consulta de datos a la base de datos. En este caso los métodos se instrumentan para obtener los datos del fichero de trazas en lugar de obtenerlos de la base de datos, de forma que se puede comprobar si la ejecución ha sido correcta.

Aplicación de prueba Para la ejecución se desarrolló una aplicación de una única clase que realiza el acceso a la base de datos y pinta los datos que obtiene por pantalla. Esta aplicación se pasa (el nombre completo) a la aplicación que instrumenta en los parámetros de entrada del main de la clase principal. La clase es la siguiente:

es.soporte.trazas.ejecutables.bbdd.Conexion (la clase de prueba)

La clase es la misma que en la ejecución, ya que intentamos comprobar la reejecución con los datos del fichero.

Resultado de la ejecución El resultado de la ejecución es el siguiente:

———————————————————————————————

Nombre: Eva | Edad: 30 | NIF: 07721443M | Disponible: 111.111 | Actualizado: true

Nombre: Maria | Edad: 15 | NIF: 12345678J | Disponible: 8571.43 | Actualizado: false

Nombre: Pedro | Edad: 32 | NIF: 12354678J | Disponible: 44166.7 | Actualizado: true

———————————————————————————————

Se puede comprobar que el resultado es el mismo que en la ejecución de la clase de forma independiente. Por tanto, la funcionalidad de la clase no ha cambiado. Esto es así porque en la instrumentación de la clase no modificamos el comportamiento, solo añadimos más (el almacenamiento de los datos en el fichero de trazas).

Trazas de la ejecución Las trazas de ejecución de la aplicación son las siguientes. En ellas se indican los métodos que se instrumentan:

Evita paquete:com.mysql.jdbc
Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.bbdd.Conexion"
Instrumentado llamada al método: "getConnection" de la clase: "java.sql.DriverManager"
Instrumentado llamada al método: "next" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getString" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getInt" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getString" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getFloat" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getBoolean" de la clase: "java.sql.ResultSet"
Instrumenta llamada a métodos de la clase: "es.pfc.log.exception.LoggerException"
Instrumenta llamada a métodos de la clase: "es.pfc.log.Logger"
Fichero: D:|Software|PFC|workspace|PFC|trazas.dat
El total de variables leídas: 19

Se indica en cada línea:

- 1 - En esta línea se indica un paquete a evitar. En este caso evita el jar del fabricante de la base de datos, donde se hace el acceso real, ya que si se instrumentan estas clases se dejaría de hacer el acceso a las tablas reales.
- 2 - En esta línea se indica que clase se instrumenta (*es.soporte.trazas.ejecutables.bbdd.Conexion*), que en nuestro caso es la clase de prueba.
- 3 a 9 - En estas líneas se indican que llamadas a métodos de la clase de prueba se instrumenta. El caso de la reejecución es un método más (*getConnection*) que se debe instrumentar para realizar una conexión ficticia a una bbdd datos local que permita ejecutar el proceso de forma correcta.

- 10 a 11 - En estas líneas se indican otras clases en las que se buscan llamadas a métodos de acceso a bbdd. En este caso son las clases del Logger, que se cargan para almacenar los datos en el fichero. Estas clases no se evitan porque en el caso de base de datos no hay acceso por parte de ellas a las tablas. Si no fuese así deberían estar en la lista de paquetes a evitar. En las trazas se comprueba que no se han encontrado llamadas a métodos de bbdd ya que no hay trazas de instrumentación de ninguna llamada.
- 12 - Path completo del fichero de trazas que se genera.
- 13 - Indica el número de variables leídas del fichero de trazas. Son los datos con los que se ejecutará el proceso.

En la imagen 7.2 se puede ver que los métodos instrumentados son los correctos. Según las trazas se hacen las siguientes llamadas a los siguientes métodos:

- getConnection()
- next()
- getString()
- getInt()
- getString()
- getFloat()
- getBoolean()

Estas llamadas son las que dicen las trazas que se instrumentan, por lo tanto la instrumentación parece correcta. Para comprobar que realmente se han instrumentado bien los métodos hay que mirar que los datos que se usan son los que se obtiene del fichero de trazas.

7.4.2.2. Reejecución a partir de un fichero creado/modificado manualmente

Para comprobar que realmente la reejecución es correcta, modificamos el fichero de trazas manualmente eliminando datos como si se hubiesen leído menos datos de los que realmente se han leído de la tabla de base de datos.

A continuación se muestran los resultados de la reejecución.

Fichero de pruebas: Datos del fichero:
0 true
1 Pedro
2 32
3 12354678J
4 44166.7

5 true
6 true
7 Eva 2
8 32
9 7721443M
10 122.111
11 true
12 false

A los datos que teníamos en el fichero (teníamos 18 datos) eliminamos los datos del último registro (6 datos)

Resultado ejecución: El resultado de la ejecución del proceso es como si en la tabla de base de datos hubiese un registro menos (el que hemos eliminado del fichero).

————————————————————————————————————

Nombre: Pedro | Edad: 32 | NIF: 12354678J | Disponible: 44166.7 | Actualizado: true

Nombre: Eva 2 | Edad: 32 | NIF: 7721443M | Disponible: 122.111 | Actualizado: true

————————————————————————————————————

Trazas de la ejecución Las trazas de ejecución son las mismas que en la reejecución. Es correcto que sea así ya que la clase de prueba usada en todos los casos es la misma..

Evita paquete:com.mysql.jdbc
Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.bbdd.Conexion"
Instrumentado llamada al método: "getConnection" de la clase: "java.sql.DriverManager"
Instrumentado llamada al método: "next" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getString" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getInt" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getString" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getFloat" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getBoolean" de la clase: "java.sql.ResultSet"
Instrumenta llamada a métodos de la clase: "es.pfc.log.exception.LoggerException"
Instrumenta llamada a métodos de la clase: "es.pfc.log.Logger"
Fichero: D:\Software\PFC\workspace\PFC\trazas.dat
El total de variables leídas: 12

Como se puede ver la instrumentación es correcta. El número de variables leídas concuerda con las del fichero. Las llanadas a métodos instrumentados también concuerdan con las llamadas reales que se hacen.

7.5. Acceso a Ficheros - FileInputStream

7.5.1. Ejecución

Aplicación de instrumentación Para instrumentar las clases de acceso a bbdd se desarrolló la siguiente clase:

es.pfc.trazas.javassist.exec.file.ObtenerTrazasInstrumentar

Esta clase carga la clase que se le pasa por parámetro, ejecuta su main e instrumenta las llamadas a los métodos de lectura de ficheros. Estos datos se almacenan en el fichero de trazas, de forma que pueden ser recuperados posteriormente para la reejecución.

Aplicación de prueba Para probar el acceso a ficheros se desarrolló una clase simple que realizaba los pasos básicos:

1. Comprueba que el fichero existe y lo abre.
2. Lee el contenido con los 3 métodos básicos (read) muestra los resultados por la salida estándar
3. Cierra el fichero

La clase se llama *es.soporte.trazas.ejecutables.file.LecturaFileInputStream* y tiene un main que comprueba y abre el fichero que se le indica y muestra el contenido del mismo.

Resultado de la ejecución El resultado de la ejecución es el siguiente:

————————————
Método: int read()
————————————
leído: 76
leído: 101
leído: 101
leído: 114
————————————
Método: int read(byte[] b)
————————————
Leído: Leer
————————————
Método: int read(byte[] b, int off, int len)
————————————
leído: Le
leído: er

Para cada uno de los métodos a instrumentar se pinta los datos como se leen. En el primer caso se leen carácter a carácter, en el segundo se lee una línea completa de golpe y en el tercer caso se lee mediante un buffer cuya longitud es escogida por el programador..

Se puede comprobar que el resultado es el mismo que en la ejecución de la clase de prueba de forma independiente. Por tanto, la funcionalidad de la clase no ha cambiado. Esto es así porque en la instrumentación de la clase no modificamos el comportamiento, solo añadimos más (el almacenamiento de los datos en el fichero de trazas).

Trazas de la ejecución Las trazas de ejecución de la aplicación son las siguientes. En ellas se indican los métodos que se instrumentan:

Evita paquete:es.pfc.log
Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.file.LecturaFileInputStream"
Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"
Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"
Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"
Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"
Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"
Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"
Fichero: D:\Software\PFC\workspace\PFC\trazas.dat
El fichero existe, lo borramos...

Se indica en cada línea:

- 1 - En esta línea se indica un paquete a evitar. En este caso evita las clases que guardan trazas (nuestro logger), ya que si se instrumentan estas clases se dejaría de guardar los valores en el fichero de trazas.
- 2 - En esta línea se indica que clase se instrumenta (*es.soporte.trazas.ejecutables.file.LecturaFileInputStrean* que en nuestro caso es la clase de prueba.
- 3 a 11 - En estas líneas se indican que llamadas a métodos de la clase de prueba se instrumenta.
- 12 - Path completo del fichero de trazas que se genera.
- 13 - Indica que el fichero ya existía de otra ejecución anterior, se elimina y se crea un fichero nuevo vacío que contendrá los datos únicamente de la ejecución actual.

En la imagen 7.3 se puede ver que los métodos instrumentados son los correctos. Según las trazas se hacen las siguientes llamadas a los siguientes métodos:

- exists()
- read()
- read()
- exists()

Figura 7.3: Clase de prueba para acceso a fichero de tipo InputStream

- read()
- read()
- exists()
- read()
- read()

Estas llamadas son las que dicen las trazas que se instrumentan, por lo tanto la instrumentación parece correcta. Para comprobar que realmente se han instrumentado bien los métodos hay que mirar que en el fichero de trazas se hayan almacenado los valores de las variables que se leen.

Contenido del fichero de trazas En el fichero de trazas tenemos el siguiente contenido:

0 true
1 76
2 101
3 101
4 114
5 -1
6 true
7 76
7 101
7 101
7 114
8 4
9 76
9 101
9 101
9 114
10 -1
11 true
12 76
12 101
12 0
12 0
13 2
14 101
14 114
14 0
14 0
15 2
16 101
16 114
16 0
16 0
17 -1

Se trata de los enteros que representan los bytes leídos del fichero (3 veces, una por cada método) y el -1 final que indica el final del contenido en cada caso (o el número de bytes leídos en los casos correspondientes).

7.5.2. Reejecución

7.5.2.1. Reejecución con el fichero generado en la ejecución

Aplicación de instrumentación Para instrumentar las clases de acceso a ficheros se desarrolló la siguiente clase:

es.pfc.trazas.javassist.exec.file.EjecutarConTrazasInstrumentar

Esta clase carga la clase que se le pasa por parámetro, ejecuta su main e instrumenta las llamadas a los métodos de consulta de datos a ficheros de tipo InputStream. En este caso los métodos se instrumentan para obtener los datos del fichero de trazas en lugar de obtenerlos del fichero original, de forma que se puede comprobar si la ejecución ha sido correcta.

Aplicación de prueba Para la ejecución se desarrolló una aplicación de una única clase que abre un fichero, lee el contenido y pinta los datos que obtiene por pantalla. Hay tres métodos de lectura de datos de un fichero de este tipo, por tanto hay tres métodos distintos para probar las tres llamadas. Esta aplicación se pasa (el nombre completo) a la aplicación que instrumenta en los parámetros de entrada del main de la clase principal. La clase es la siguiente:

es.soporte.trazas.ejecutables.file.LecturaFileInputStream(la clase de prueba)

La clase es la misma que en la ejecución, ya que intentamos comprobar la reejecución con los datos del fichero.

Resultado de la ejecución El resultado de la ejecución es el siguiente:

———————————
Método: int read()
———————————
leído: 76
leído: 101
leído: 101
leído: 114
———————————
Método: int read(byte[] b)
———————————
Leído: Leer
———————————
Método: int read(byte[] b, int off, int len)
———————————
leído: Le
leído: er

Se puede comprobar que el resultado es el mismo que en la ejecución de la clase de forma independiente. Por tanto, la funcionalidad de la clase no ha cambiado. Esto es así porque en la instrumentación de la clase no modificamos el comportamiento, solo añadimos más (el almacenamiento de los datos en el fichero de trazas).

Trazas de la ejecución Las trazas de ejecución de la aplicación son las siguientes. En ellas se indican los métodos que se instrumentan:

Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.file.LecturaFileInputStream"

Instrumentado llamada al método: "exists" de la clase: "java.io.File"

Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"

Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"

Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"

Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"

Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"

Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"

Instrumenta llamada a métodos de la clase: "es.pfc.log.exception.LoggerException"

Fichero: D:\Software\PFC\workspace\PFC\trazas.dat
El total de variables leídas: 21
Se indica en cada línea:

- 1 - En esta línea se indica que clase se instrumenta (*es.soporte.trazas.ejecutables.bbdd.Conexiones.soporte.t*
que en nuestro caso es la clase de prueba.
- 2 a 10 - En estas líneas se indican que llamadas a métodos de la clase de prueba se instrumenta.
- 11 - En estas líneas se indican otras clases en las que se buscan llamadas a métodos de acceso a ficheros. En este caso son las clases del Logger, que se cargan para almacenar los datos en el fichero. En las trazas se comprueba que no se han encontrado llamadas a métodos de acceso a ficheros ya que no hay trazas de instrumentación de ninguna llamada.
- 12 - Path completo del fichero de trazas que se genera.
- 13 - Indica el número de variables leídas del fichero de trazas. Son los datos con los que se ejecutará el proceso.

En la imagen 7.3 se puede ver que los métodos instrumentados son los correctos. Según las trazas se hacen las siguientes llamadas a los siguientes métodos:

- exists()
- read()
- read()
- exists()
- read()

- read()
- exists()
- read()
- read()

Estas llamadas son las que dicen las trazas que se instrumentan, por lo tanto la instrumentación parece correcta. Para comprobar que realmente se han instrumentado bien los métodos hay que mirar que los datos que se usan son los que se obtiene del fichero de trazas.

7.5.2.2. Reejecución a partir de un fichero creado/modificado manualmente

Para comprobar que realmente la reejecución es correcta, modificamos el fichero de trazas manualmente eliminando datos como si se hubiesen leído menos datos de los que realmente se han leído del fichero de prueba.

A continuación se muestran los resultados de la reejecución.

Fichero de pruebas: Datos del fichero:

0 true
1 76
2 101
3 101
4 114
5 -1
6true
7 76
7 101
7 101
7 114
8 2
9 76
9 101
10 -1
11 true
12 76
12 101
12 0
12 0
13 1
14 101
14 0
14 0
14 0

15 1
16 101
16 0
16 0
16 0
17 -1

A los datos que teníamos en el fichero (teníamos 18 datos) eliminamos dos letras de cada método.

Resultado ejecución: El resultado de la ejecución del proceso es como si hubiese menos letras en el fichero (las que hemos eliminado del fichero).

——————————
Método: int read()
——————————
leído: 76
leído: 101
——————————
Método: int read(byte[] b)
——————————
Leído: Le
——————————
Método: int read(byte[] b, int off, int len)
——————————
leído: L
leído: e

Trazas de la ejecución Las trazas de ejecución son las mismas que en la reejecución. Es correcto que sea así ya que la clase de prueba usada en todos los casos es la misma..

Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.file.LecturaFileInputStream"
Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"
Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"
Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"
Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"
Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"
Instrumentado llamada al método: "read" de la clase: "java.io.FileInputStream"
Instrumenta llamada a métodos de la clase: "es.pfc.log.exception.LoggerException"

Fichero: D:\Software\PFC\workspace\PFC\trazas.dat
El total de variables leídas: 16

Como se puede ver la instrumentación es correcta. El número de variables leídas concuerda con las del fichero. Las llanadas a métodos instrumentados también concuerdan con las llamadas reales que se hacen.

7.6. Acceso a Ficheros - FileReader

7.6.1. Ejecución

Aplicación de instrumentación Para instrumentar las clases de acceso a bbdd se desarrolló la siguiente clase:

es.pfc.trazas.javassist.exec.file.ObtenerTrazasInstrumentar

Esta clase carga la clase que se le pasa por parámetro, ejecuta su main e instrumenta las llamadas a los métodos de lectura de ficheros. Estos datos se almacenan en el fichero de trazas, de forma que pueden ser recuperados posteriormente para la reejecución.

Aplicación de prueba Para probar el acceso a ficheros se desarrolló una clase simple que realizaba los pasos básicos:

1. Comprueba que el fichero existe y lo abre.
2. Lee el contenido y lo muestra por la salida estándar
3. Cierra el fichero

La clase se llama *es.soporte.trazas.ejecutables.file.LecturaFicheroFileReader* y tiene un main que comprueba y abre el fichero que se le indica y muestra el contenido del mismo.

Resultado de la ejecución El resultado de la ejecución de esta aplicación es el siguiente:

——————————

Leer

——————————

El proceso abre el fichero, lee el contenido y lo pinta por pantalla.

Se puede comprobar que el resultado es el mismo que en la ejecución de la clase de prueba de forma independiente. Por tanto, la funcionalidad de la clase no ha cambiado. Esto es así porque en la instrumentación de la clase no modificamos el comportamiento, solo añadimos más (el almacenamiento de los datos en el fichero de trazas).

Trazas de la ejecución Las trazas de ejecución de la aplicación son las siguientes. En ellas se indican los métodos que se instrumentan:

Evita paquete:es.pfc.log
Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.file.LecturaFicheroFileReader"
Instrumentado llamada al método: "exists" de la clase: "java.io.File"

Instrumentado llamada al método: "read" de la clase: "java.io.FileReader"
Instrumentado llamada al método: "read" de la clase: "java.io.FileReader"
Fichero: D:\Software\PFC\workspace\PFC\trazas.dat
El fichero existe, lo borramos...
Se indica en cada línea:

- 1 - En esta línea se indica un paquete a evitar. En este caso evita las clases que guardan trazas (nuestro logger), ya que si se instrumentan estas clases se dejaría de guardar los valores en el fichero de trazas.
- 2 - En esta línea se indica que clase se instrumenta (*es.soporte.trazas.ejecutables.file.LecturaFicheroFileRea* que en nuestro caso es la clase de prueba.
- 3 a 5 - En estas líneas se indican que llamadas a métodos de la clase de prueba se instrumenta.
- 6 - Path completo del fichero de trazas que se genera.
- 7 - Indica que el fichero ya existía de otra ejecución anterior, se elimina y se crea un fichero nuevo vacío que contendrá los datos únicamente de la ejecución actual.

En la imagen 7.4 se puede ver que los métodos instrumentados son los correctos. Según las trazas se hacen las siguientes llamadas a los siguientes métodos:

- exists()
- read()
- read()

Estas llamadas son las que dicen las trazas que se instrumentan, por lo tanto la instrumentación parece correcta. Para comprobar que realmente se han instrumentado bien los métodos hay que mirar que en el fichero de trazas se hayan almacenado los valores de las variables que se leen.

Contenido del fichero de trazas En el fichero de trazas tenemos el siguiente contenido:
0 true
1 76
2 101
3 101
4 114
5 -1
Se trata de los enteros que representan los bytes leídos del fichero y el -1 final que indica el final del contenido.

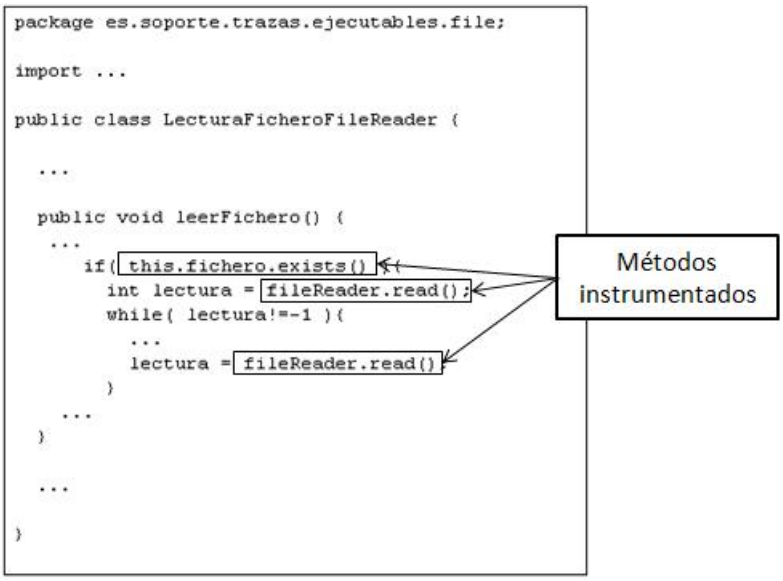


Figura 7.4: Clase de prueba para acceso a fichero de tipo Reader

7.6.2. Reejecución

7.6.2.1. Reejecución con el fichero generado en la ejecución

Aplicación de instrumentación Para instrumentar las clases de acceso a ficheros se desarrolló la siguiente clase:

es.pfc.trazas.javassist.exec.file.EjecutarConTrazasInstrumentar

Esta clase carga la clase que se le pasa por parámetro, ejecuta su main e instrumenta las llamadas a los métodos de consulta de datos a ficheros de tipo InputStream. En este caso los métodos se instrumentan para obtener los datos del fichero de trazas en lugar de obtenerlos del fichero original, de forma que se puede comprobar si la ejecución ha sido correcta.

Aplicación de prueba Para la ejecución se desarrolló una aplicación de una única clase que abre un fichero, lee el contenido y pinta los datos que obtiene por pantalla. Esta aplicación se pasa (el nombre completo) a la aplicación que instrumenta en los parámetros de entrada del main de la clase principal. La clase es la siguiente:

es.pfc.trazas.clasessoporte.LecturaFicheroFileReader (la clase de prueba)

La clase es la misma que en la ejecución, ya que intentamos comprobar la reejecución con los datos del fichero.

Resultado de la ejecución El resultado de la ejecución es el siguiente:

——————————

Leer
————————————

Se puede comprobar que el resultado es el mismo que en la ejecución de la clase de forma independiente. Por tanto, la funcionalidad de la clase no ha cambiado. Esto es así porque en la instrumentación de la clase no modificamos el comportamiento, solo añadimos más (el almacenamiento de los datos en el fichero de trazas).

Trazas de la ejecución Las trazas de ejecución de la aplicación son las siguientes. En ellas se indican los métodos que se instrumentan:

Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.file.LecturaFicheroFileReader"
Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "read" de la clase: "java.io.FileReader"
Instrumentado llamada al método: "read" de la clase: "java.io.FileReader"
Instrumenta llamada a métodos de la clase: "es.pfc.log.exception.LoggerException"
Fichero: D:\Software\PFC\workspace\PFC\trazas.dat
El total de variables leídas: 6
Se indica en cada línea:

- 1 - En esta línea se indica que clase se instrumenta (*es.soporte.trazas.ejecutables.file.LecturaFicheroFileRea*
que en nuestro caso es la clase de prueba.
- 2 a 4 - En estas líneas se indican que llamadas a métodos de la clase de prueba se instrumenta.
- 5 - En estas líneas se indican otras clases en las que se buscan llamadas a métodos de acceso a ficheros. En este caso son las clases del Logger, que se cargan para almacenar los datos en el fichero. En las trazas se comprueba que no se han encontrado llamadas a métodos de bbdd ya que no hay trazas de instrumentación de ninguna llamada.
- 6 - Path completo del fichero de trazas que se genera.
- 7 - Indica el número de variables leídas del fichero de trazas. Son los datos con los que se ejecutará el proceso.

En la imagen 7.4 se puede ver que los métodos instrumentados son los correctos. Según las trazas se hacen las siguientes llamadas a los siguientes métodos:

- exists()
- read()
- read()

Estas llamadas son las que dicen las trazas que se instrumentan, por lo tanto la instrumentación parece correcta. Para comprobar que realmente se han instrumentado bien los métodos hay que mirar que los datos que se usan son los que se obtiene del fichero de trazas.

7.6.2.2. Reejecución a partir de un fichero creado/modificado manualmente

Para comprobar que realmente la reejecución es correcta, modificamos el fichero de trazas manualmente modificando los datos como si se hubiesen leído otros datos que los que realmente se han leído del fichero de prueba.

A continuación se muestran los resultados de la reejecución.

Fichero de pruebas: Datos del fichero:

0 true
1 76
2 101
3 101
4 114
5 76
6 101
7 101
8 114
9 -1

A los datos que teníamos en el fichero añadimos el texto leído de nuevo.

Resultado ejecución: El resultado de la ejecución del proceso es como si el texto estuviese repetido.

——————————
LeerLeer
——————————

Trazas de la ejecución Las trazas de ejecución son las mismas que en la reejecución. Es correcto que sea así ya que la clase de prueba usada en todos los casos es la misma..

Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.file.LecturaFicheroFileReader"
Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "read" de la clase: "java.io.FileReader"
Instrumentado llamada al método: "read" de la clase: "java.io.FileReader"
Instrumenta llamada a métodos de la clase: "es.pfc.log.exception.LoggerException"
Fichero: D:\Software\PFC\workspace\PFC\trazas.dat
El total de variables leídas: 10

Como se puede ver la instrumentación es correcta. El número de variables leídas concuerda con las del fichero. Las llanadas a métodos instrumentados también concuerdan con las llamadas reales que se hacen.

7.7. Acceso a Ficheros - BufferedReader

7.7.1. Ejecución

Aplicación de instrumentación Para instrumentar las clases de acceso a bbdd se desarrolló la siguiente clase:

es.pfc.trazas.javassist.exec.file.ObtenerTrazasInstrumentar

Esta clase carga la clase que se le pasa por parámetro, ejecuta su main e instrumenta las llamadas a los métodos de lectura de ficheros. Estos datos se almacenan en el fichero de trazas, de forma que pueden ser recuperados posteriormente para la reejecución.

Aplicación de prueba Para probar el acceso a ficheros se desarrolló una clase simple que realizaba los pasos básicos:

1. Comprueba que el fichero existe y lo abre.
2. Lee el contenido y lo muestra por la salida estándar
3. Cierra el fichero

La clase se llama *es.pfc.trazas.clasessoporte.file.LecturaFicheroBufferedReader* y tiene un main que comprueba que existe y abre el fichero que se le indica y muestra el contenido del mismo.

Resultado de la ejecución El resultado de la ejecución de esta aplicación es el siguiente:

——————————

Leer

——————————

El proceso abre el fichero, lee el contenido y lo pinta por pantalla.

Se puede comprobar que el resultado es el mismo que en la ejecución de la clase de prueba de forma independiente. Por tanto, la funcionalidad de la clase no ha cambiado. Esto es así porque en la instrumentación de la clase no modificamos el comportamiento, solo añadimos más (el almacenamiento de los datos en el fichero de trazas).

Trazas de la ejecución Las trazas de ejecución de la aplicación son las siguientes. En ellas se indican los métodos que se instrumentan:

Evita paquete:es.pfc.log
Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.file.LecturaFicheroBufferedReader"
Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "readLine" de la clase: "java.io.BufferedReader"
Instrumentado llamada al método: "readLine" de la clase: "java.io.BufferedReader"
Fichero: D:|Software|PFC|workspace|PFC|trazas.dat
El fichero existe, lo borramos....

Se indica en cada línea:

- 1 - En esta línea se indica un paquete a evitar. En este caso evita las clases que guardan trazas (nuestro logger), ya que si se instrumentan estas clases se dejaría de guardar los valores en el fichero de trazas.
- 2 - En esta línea se indica que clase se instrumenta (*es.soporte.trazas.ejecutables.file.LecturaFicheroBuffere*
que en nuestro caso es la clase de prueba.
- 3 a 5 - En estas líneas se indican que llamadas a métodos de la clase de prueba se instrumenta.
- 6 - Path completo del fichero de trazas que se genera.
- 7 - Indica que el fichero ya existía de otra ejecución anterior, se elimina y se crea un fichero nuevo vacío que contendrá los datos únicamente de la ejecución actual.

En la imagen 7.5 se puede ver que los métodos instrumentados son los correctos. Según las trazas se hacen las siguientes llamadas a los siguientes métodos:

- exists()
- read()
- read()

Estas llamadas son las que dicen las trazas que se instrumentan, por lo tanto la instrumentación parece correcta. Para comprobar que realmente se han instrumentado bien los métodos hay que mirar que en el fichero de trazas se hayan almacenado los valores de las variables que se leen.

Contenido del fichero de trazas En el fichero de trazas tenemos el siguiente contenido:

0 true
1 Leer
2 null

Se trata del flag que indica que el fichero existe, la lectura del contenido y el valor null final que indica el fin de fichero.

7.7.2. Reejecución

7.7.2.1. Reejecución con el fichero generado en la ejecución

Aplicación de instrumentación Para instrumentar las clases de acceso a ficheros se desarrolló la siguiente clase:

es.pfc.trazas.javassist.exec.file.EjecutarConTrazasInstrumentar

Esta clase carga la clase que se le pasa por parámetro, ejecuta su main e instrumenta las llamadas a los métodos de consulta de datos a ficheros de tipo InputStream. En este caso los métodos se instrumentan para obtener los datos del fichero de trazas en lugar de obtenerlos del fichero original, de forma que se puede comprobar si la ejecución ha sido correcta.

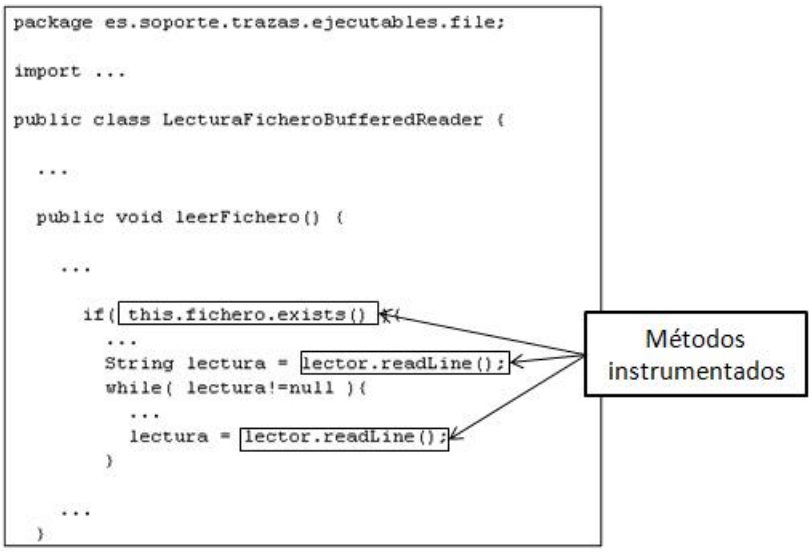


Figura 7.5: Clase de prueba para acceso a fichero de tipo BufferedReader

Aplicación de prueba Para la ejecución se desarrolló una aplicación de una única clase que abre un fichero, lee el contenido y pinta los datos que obtiene por pantalla. Esta aplicación se pasa (el nombre completo) a la aplicación que instrumenta en los parámetros de entrada del main de la clase principal. La clase es la siguiente:

es.pfc.trazas.clasessoporte.LecturaFicheroFileReader (la clase de prueba)

La clase es la misma que en la ejecución, ya que intentamos comprobar la reejecución con los datos del fichero.

Resultado de la ejecución El resultado de la ejecución es el siguiente:

——————————

Leer

——————————

Se puede comprobar que el resultado es el mismo que en la ejecución de la clase de forma independiente. Por tanto, la funcionalidad de la clase no ha cambiado. Esto es así porque en la instrumentación de la clase no modificamos el comportamiento, solo añadimos más (el almacenamiento de los datos en el fichero de trazas).

Trazas de la ejecución Las trazas de ejecución de la aplicación son las siguientes. En ellas se indican los métodos que se instrumentan:

Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.file.LecturaFicheroBufferedReader"
Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "readLine" de la clase: "java.io.BufferedReader"

Instrumentado llamada al método: "readLine" de la clase: "java.io.BufferedReader"
Instrumenta llamada a métodos de la clase: "es.pfc.log.exception.LoggerException"
Fichero: D:\Software\PFC\workspace\PFC\trazas.dat
El total de variables leídas: 3
Se indica en cada línea:

- 1 - En esta línea se indica que clase se instrumenta (*es.soporte.trazas.ejecutables.file.LecturaFicheroBuffere*
que en nuestro caso es la clase de prueba.
- 2 a 4 - En estas líneas se indican que llamadas a métodos de la clase de prueba se instrumenta.
- 5 - En estas líneas se indican otras clases en las que se buscan llamadas a métodos de acceso a ficheros. En este caso son las clases del Logger, que se cargan para almacenar los datos en el fichero. En las trazas se comprueba que no se han encontrado llamadas a métodos de acceso a ficheros ya que no hay trazas de instrumentación de ninguna llamada.
- 6 - Path completo del fichero de trazas que se genera.
- 7 - Indica el número de variables leídas del fichero de trazas. Son los datos con los que se ejecutará el proceso.

En la imagen 7.5 se puede ver que los métodos instrumentados son los correctos. Según las trazas se hacen las siguientes llamadas a los siguientes métodos:

- exists()
- read()
- read()

Estas llamadas son las que dicen las trazas que se instrumentan, por lo tanto la instrumentación parece correcta. Para comprobar que realmente se han instrumentado bien los métodos hay que mirar que los datos que se usan son los que se obtiene del fichero de trazas.

En el caso de no tener el fichero en el directorio (que seria el caso real, donde solo dispondríamos del log de trazas) tendríamos un problema, ya que al ir a comprobar si existe da un error en la aplicación:

Clase: es.pfc.trazas.clasessoporte.file.LecturaFicheroBufferedReader
Instrumentado: Clase: java.io.File – método: exists
Instrumentado: Clase: java.io.BufferedReader – método: readLine
Instrumentado: Clase: java.io.BufferedReader – método: readLine
Fichero: D:\Software\PFC\workspace\PFC\trazas.dat
java.io.FileNotFoundException: d:\pfc\pruebas\lectura.TXT (El sistema no puede encontrar el archivo especificado)
at java.io.FileInputStream.open(Native Method)
at java.io.FileInputStream.<init>(Unknown Source)
at java.io.FileReader.<init>(Unknown Source)
at es.pfc.trazas.clasessoporte.file.LecturaFicheroBufferedReader.leerFichero(LecturaFicheroBufferedReader.java:39)
at es.pfc.trazas.clasessoporte.file.LecturaFicheroBufferedReader.main(LecturaFicheroBufferedReader.java:77)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
at java.lang.reflect.Method.invoke(Unknown Source) at javassist.Loader.run(Loader.java:290)
at es.pfc.trazas.javassist.file.EjecutarConTrazasInstrumentar.main(EjecutarConTrazasInstrumentar.java:35)

Este problemas se tendría con muchos otros métodos que proporciona la clase y que dependen de la existencia del fichero físico.

7.7.2.2. Reejecución a partir de un fichero creado/modificado manualmente

Para comprobar que realmente la reejecución es correcta, modificamos el fichero de trazas manualmente cambiando los datos como si se hubiesen leído otros datos que los que realmente se han leído del fichero de prueba.

A continuación se muestran los resultados de la reejecución.

Fichero de pruebas: Datos del fichero:

0 true
1 Leer
2 Leer
3 Leer
4 null

A los datos que teníamos en el fichero añadimos el texto leído dos veces más.

Resultado ejecución: El resultado de la ejecución del proceso es como si el texto estuviese repetido las veces que lo hemos puesto en el fichero de trazas.

———————————
Leer
Leer
Leer
———————————

Trazas de la ejecución Las trazas de ejecución son las mismas que en la reejecución. Es correcto que sea así ya que la clase de prueba usada en todos los casos es la misma..

Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.file.LecturaFicheroBufferedReader"
Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "readLine" de la clase: "java.io.BufferedReader"
Instrumentado llamada al método: "readLine" de la clase: "java.io.BufferedReader"
Instrumenta llamada a métodos de la clase: "es.pfc.log.exception.LoggerException"
Fichero: D:\Software\PFC\workspace\PFC\trazas.dat
El total de variables leídas: 5

Como se puede ver la instrumentación es correcta. El número de variables leídas concuerda con las del fichero. Las llanadas a métodos instrumentados también concuerdan con las llamadas reales que se hacen.

7.8. Acceso a Ficheros - DataInputStream

7.8.1. Ejecución

Aplicación de instrumentación Para instrumentar las clases de acceso a bbdd se desarrolló la siguiente clase:

es.pfc.trazas.javassist.exec.file.ObtenerTrazasInstrumentar

Esta clase carga la clase que se le pasa por parámetro, ejecuta su main e instrumenta las llamadas a los métodos de lectura de ficheros. Estos datos se almacenan en el fichero de trazas, de forma que pueden ser recuperados posteriormente para la reejecución.

Aplicación de prueba Las clases derivadas de DataInput leen datos primitivos de Java. Para probar el proceso de instrumentación se creó un fichero con datos primitivos Java y una clase que leía estos datos (mediante el acceso a base de datos y los datos de entrada no había sido posible comprobar todos los tipos básicos de datos).

La clase de prueba generada realiza las siguientes acciones sobre el fichero de prueba:

1. Comprueba que el fichero existe y lo abre.
2. Lee el contenido y lo muestra por la salida estándar
3. Cierra el fichero

La clase se llama *es.soporte.trazas.ejecutables.file.LecturaDataInput* y tiene un main que comprueba que existe y abre el fichero que se le indica y muestra el contenido del mismo.

Resultado de la ejecución El resultado de la ejecución de esta aplicación es el siguiente:

———————————

boolean: false
byte: 122
char: A
double: 1.246757290217659E37
float: 1.2467573E37
int: 65345
long: 1125899906842624
short: 8192

———————————

El proceso abre el fichero, lee el contenido y lo pinta por pantalla.

Se puede comprobar que el resultado es el mismo que en la ejecución de la clase de prueba de forma independiente. Por tanto, la funcionalidad de la clase no ha cambiado. Esto es así porque en la instrumentación de la clase no modificamos el comportamiento, solo añadimos más (el almacenamiento de los datos en el fichero de trazas).

Trazas de la ejecución Las trazas de ejecución de la aplicación son las siguientes. En ellas se indican los métodos que se instrumentan:

Evita paquete:es.pfc.log
Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.file.LecturaDataInput"
Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "readBoolean" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readByte" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readChar" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readDouble" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readFloat" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readInt" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readLong" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readShort" de la clase: "java.io.DataInputStream"
Fichero: D:\Software\PFC\workspace\PFC\trazas.dat
El fichero existe, lo borramos...
Se indica en cada línea:

- 1 - En esta línea se indica un paquete a evitar. En este caso evita las clases que guardan trazas (nuestro logger), ya que si se instrumentan estas clases se dejaría de guardar los valores en el fichero de trazas.
- 2 - En esta línea se indica que clase se instrumenta (*es.soporte.trazas.ejecutables.file.LecturaDataInput*), que en nuestro caso es la clase de prueba.
- 3 a 11 - En estas líneas se indican que llamadas a métodos de la clase de prueba se instrumenta.
- 12 - Path completo del fichero de trazas que se genera.
- 13 - Indica que el fichero ya existía de otra ejecución anterior, se elimina y se crea un fichero nuevo vacío que contendrá los datos únicamente de la ejecución actual.

En la imagen 7.6 se puede ver que los métodos instrumentados son los correctos. Según las trazas se hacen las siguientes llamadas a los siguientes métodos:

- exists()
- readBoolean()
- readByte()
- readChar()
- readDouble()
- readFloat()
- readInt()
- readLong()

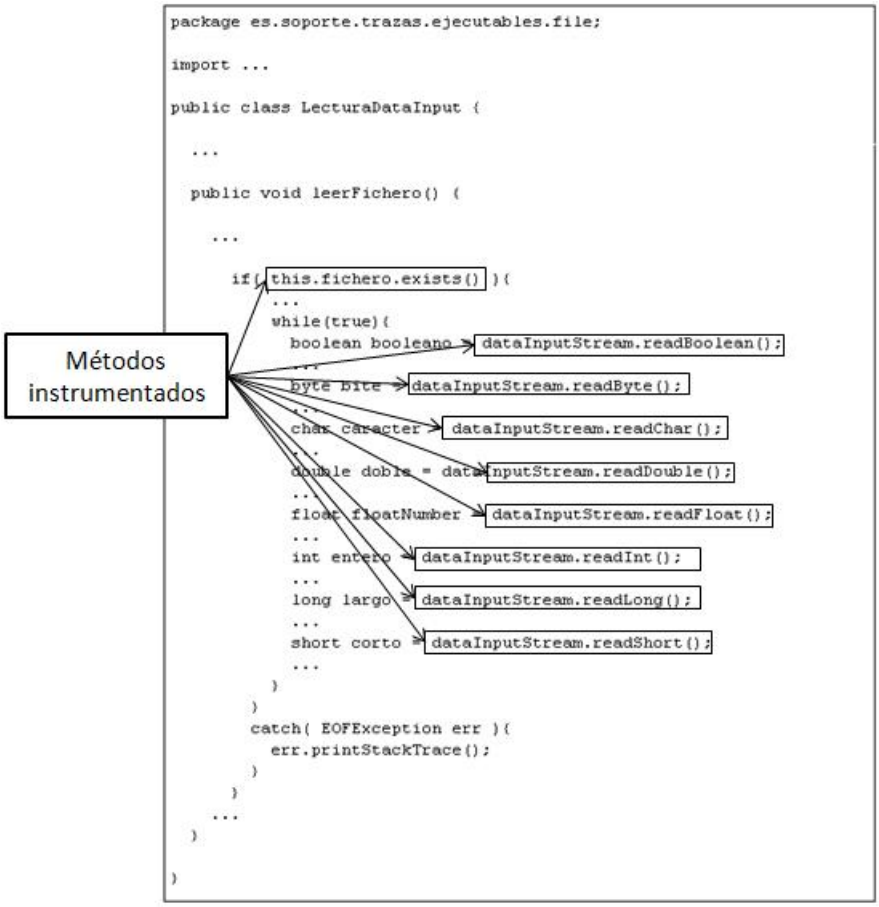


Figura 7.6: Clase de prueba para acceso a fichero de tipo DatInput

- readShort()

Estas llamadas son las que dicen las trazas que se instrumentan, por lo tanto la instrumentación parece correcta. Para comprobar que realmente se han instrumentado bien los métodos hay que mirar que en el fichero de trazas se hayan almacenado los valores de las variables que se leen.

Contenido del fichero de trazas En el fichero de trazas tenemos el siguiente contenido:

0 true
1 false
2 122
3 A
4 1.246757290217659E37

5 1.2467573E37
6 65345
7 1125899906842624
8 8192
Se trata del flag que indica que el fichero existe y la lectura del contenido.

7.8.2. Reejecución

7.8.2.1. Reejecución con el fichero generado en la ejecución

Aplicación de instrumentación Para instrumentar las clases de acceso a ficheros se desarrolló la siguiente clase:

es.pfc.trazas.javassist.exec.file.EjecutarConTrazasInstrumentar

Esta clase carga la clase que se le pasa por parámetro, ejecuta su main e instrumenta las llamadas a los métodos de consulta de datos a ficheros de tipo DataInput. En este caso los métodos se instrumentan para obtener los datos del fichero de trazas en lugar de obtenerlos del fichero original, de forma que se puede comprobar si la ejecución ha sido correcta.

Aplicación de prueba Para la ejecución se desarrolló una aplicación de una única clase que abre un fichero, lee el contenido y pinta los datos que obtiene por pantalla. Esta aplicación se pasa (el nombre completo) a la aplicación que instrumenta en los parámetros de entrada del main de la clase principal. La clase es la siguiente:

es.pfc.trazas.clasessoporte.LecturaDataInput(la clase de prueba)

La clase es la misma que en la ejecución, ya que intentamos comprobar la reejecución con los datos del fichero.

Resultado de la ejecución El resultado de la ejecución es el siguiente:

———————————
boolean: false
byte: 122
char: A
double: 1.246757290217659E37
float: 1.2467573E37
int: 65345
long: 1125899906842624
short: 8192
———————————

Se puede comprobar que el resultado es el mismo que en la ejecución de la clase de forma independiente. Por tanto, la funcionalidad de la clase no ha cambiado. Esto es así porque en la instrumentación de la clase no modificamos el comportamiento, solo añadimos más (el almacenamiento de los datos en el fichero de trazas).

Trazas de la ejecución Las trazas de ejecución de la aplicación son las siguientes. En ellas se indican los métodos que se instrumentan:

Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.file.LecturaDataInput"
Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "readBoolean" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readByte" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readChar" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readDouble" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readFloat" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readInt" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readLong" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readShort" de la clase: "java.io.DataInputStream"
Instrumenta llamada a métodos de la clase: "es.pfc.log.exception.LoggerException"
Fichero: D:\Software\PFC\workspace\PFC\trazas.dat
El total de variables leídas: 9
Se indica en cada línea:

- 1 - En esta línea se indica que clase se instrumenta (*es.soporte.trazas.ejecutables.file.LecturaDataInput*), que en nuestro caso es la clase de prueba.
- 2 a 10 - En estas líneas se indican que llamadas a métodos de la clase de prueba se instrumenta.
- 11 - En estas líneas se indican otras clases en las que se buscan llamadas a métodos de acceso a ficheros. En este caso son las clases del Logger, que se cargan para almacenar los datos en el fichero. En las trazas se comprueba que no se han encontrado llamadas a métodos de acceso a ficheros ya que no hay trazas de instrumentación de ninguna llamada.
- 12 - Path completo del fichero de trazas que se genera.
- 13- Indica el número de variables leídas del fichero de trazas. Son los datos con los que se ejecutará el proceso.

En la imagen 7.6 se puede ver que los métodos instrumentados son los correctos. Según las trazas se hacen las siguientes llamadas a los siguientes métodos:

- exists()
- readBoolean()
- readByte()
- readChar()
- readDouble()
- readFloat()
- readInt()

- readLong()
- readShort()

Estas llamadas son las que dicen las trazas que se instrumentan, por lo tanto la instrumentación parece correcta. Para comprobar que realmente se han instrumentado bien los métodos hay que mirar que los datos que se usan son los que se obtiene del fichero de trazas.

7.8.2.2. Reejecución a partir de un fichero creado/modificado manualmente

En este caso, para que la ejecución del proceso sea correcta, al modificar el proceso se ha de modificar la clase que lee. Esto es porque en la clase se lee el dato sabiendo de antemano el tipo de dato que se está leyendo. Modificamos el proceso y el fichero para obtener un dato más de tipo booleano.

Fichero de pruebas: Datos del fichero:
0 true
1 false
2 122
3 A
4 1.246757290217659E37
5 1.2467573E37
6 65345
7 1125899906842624
8 8192
9 false
A los datos que teníamos en el fichero añadimos un último valor de tipo booleano.

Resultado ejecución: El resultado de la ejecución del proceso es como si el texto tuviese un dato más de tipo booleano en el fichero de trazas.
——————————
boolean: false
byte: 122
char: A
double: 1.246757290217659E37
float: 1.2467573E37
int: 65345
long: 1125899906842624
short: 8192
boolean: false

Trazas de la ejecución Las trazas de ejecución son las mismas que en la reejecución, mas una línea más del dato de tipo booleano que hemos añadido al proceso de prueba.

Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.file.LecturaDataInput"
Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "readBoolean" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readByte" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readChar" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readDouble" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readFloat" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readInt" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readLong" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readShort" de la clase: "java.io.DataInputStream"
Instrumentado llamada al método: "readBoolean" de la clase: "java.io.DataInputStream"
Instrumenta llamada a métodos de la clase: "es.pfc.log.exception.LoggerException"
Fichero: D:|Software|PFC|workspace|PFC|trazas.dat
El total de variables leídas: 10

Como se puede ver la instrumentación es correcta. El número de variables leídas concuerda con las del fichero. Las llanadas a métodos instrumentados también concuerdan con las llamadas reales que se hacen.

7.9. Aplicaciones completas

7.9.1. Ejecución

Aplicación de instrumentación Para instrumentar las clases de acceso a bbdd se desarrolló la siguiente clase:

es.pfc.trazas.javassist.exec.all.ObtenerTrazasInstrumentar

Esta clase carga la clase que se le pasa por parámetro, ejecuta su main e instrumenta las llamadas a los métodos de lectura de ficheros. Estos datos se almacenan en el fichero de trazas, de forma que pueden ser recuperados posteriormente para la reejecución.

Aplicación de prueba En este caso se creó una aplicación compuesta por diversas clases con las que se realizan tanto accesos a la base de datos como a ficheros de datos almacenados en el filesystem del sistema. También recibe parámetros de entrada al main.

La clase de prueba generada realiza las siguientes acciones sobre el fichero de prueba:

1. Obtiene los datos de entrada de la aplicación. Espera un path a un fichero que existe.

2. Abre el fichero que se le pasa por parámetro y lo lee. El contenido del fichero es un conjunto de NIFs, cada uno en una línea del mismo.

3. Para cada NIF que lee en el fichero, consulta la tabla de personas de la bbdd, obtiene los datos y los pinta por pantalla.

La clase principal se llama *es.soporte.trazas.ejecutables.all.ClaseConTodo*.

Resultado de la ejecución El resultado de la ejecución de esta aplicación es el siguiente:

Leemos fichero: d:|pfc|pruebas|all|NIFs.TXT
————————————
NIF: 07721443M
Nombre: Eva | Edad: 30 | NIF: 07721443M | Disponible: 111.111 | Actualizado: true
————————————
————————————
NIF: 79271483M
No hay personas en la base de datos con NIF: 79271483M
————————————

El proceso abre el fichero que se le pasa por parámetro y lee el contenido. Cada línea del fichero la interpreta como un NIF y consulta los datos en la tabla de personas. Si esta existe en la tabla, pinta los datos obtenidos por pantalla, y si no existe lo indica.

Se puede comprobar que el resultado es el mismo que en la ejecución de la clase de prueba de forma independiente. Por tanto, la funcionalidad de la clase no ha cambiado. Esto es así porque en la instrumentación de la clase no modificamos el comportamiento, solo añadimos más (el almacenamiento de los datos en el fichero de trazas).

Trazas de la ejecución Las trazas de ejecución de la aplicación son las siguientes. En ellas se indican los métodos que se instrumentan:

Evita paquete:com.mysql.jdbc
Evita paquete:es.pfc.log
Instrumentamos método "main" de la clase ""
Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.all.ClaseConTodo"
Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "readLine" de la clase: "java.io.BufferedReader"
Instrumentado llamada al método: "readLine" de la clase: "java.io.BufferedReader"

Fichero: D:|Software|PFC|workspace|PFC|trazas.dat
El fichero existe, lo borramos...
Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.bbdd.DatosPersonas"
Instrumentado llamada al método: "next" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "next" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getString" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getInt" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getString" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getFloat" de la clase: "java.sql.ResultSet"

Instrumentado llamada al método: "getBoolean" de la clase: "java.sql.ResultSet"
Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.bbdd.DatosConexion"
Se indica en cada línea:

- 1 y 2 - En esta línea se indica un paquetes a evitar. En este caso evitan las clases que guardan trazas (nuestro logger), ya que si se instrumentan estas clases se dejaría de guardar los valores en el fichero de trazas. También se evitan las clases del propietario de la base de datos, ya que si se instrumentasen no se haría el acceso a la tablas.
- 2 - En esta línea se indica que se instrumenta el main de la clase principal.
- 3 - En esta línea se indica la primera clase que se instrumenta (*es.soporte.trazas.ejecutables.all.ClaseConTo*
que en nuestro caso es la clase de prueba que tiene el main.
- 4 a 6 - En estas líneas se indican que llamadas a métodos de la clase anterior se instrumentan.
- 7 - Path completo del fichero de trazas que se genera.
- 8 - Indica que el fichero ya existía de otra ejecución anterior, se elimina y se crea un fichero nuevo vacío que contendrá los datos únicamente de la ejecución actual.
- 9 - En esta línea se indica la segunda clase que se instrumenta (*es.soporte.trazas.ejecutables.bbdd.DatosPers*
que en nuestro caso es la siguiente clase que se carga.
- 10 a 18 - En estas líneas se indican que llamadas a métodos de la clase anterior se instrumentan.
- 19 - En esta línea se indica la tercera clase que se instrumenta (*es.soporte.trazas.ejecutables.bbdd.DatosCon*
que en nuestro caso es la siguiente clase que se carga. En esta clase no existen llamadas a métodos para instrumentar, por tanto no hay más trazas de ejecución.

En la imagen 7.7 se puede ver que los métodos instrumentados son los correctos. Según las trazas se hacen las siguientes llamadas a los siguientes métodos:

- exists()
- readLine()
- getString()
- getInt()
- getString()
- getBoolean()
- getFloat()

- next()
- next()

Estas llamadas son las que dicen las trazas que se instrumentan, por lo tanto la instrumentación parece correcta. Para comprobar que realmente se han instrumentado bien los métodos hay que mirar que en el fichero de trazas se hayan almacenado los valores de las variables que se leen.

Contenido del fichero de trazas En el fichero de trazas tenemos el siguiente contenido:

0 d:|pfc|pruebas|all|NIFs.TXT
1 true
2 07721443M
3 true
4 Eva
5 30
6 07721443M
7 111.111
8 true
9 false
10 79271483M
11 false
12 false
13 null

Se trata del fichero de nifs a leer y los resultados de la consulta a bbdd con los nifs leídos del fichero.

7.9.2. Reejecución

7.9.2.1. Reejecución con el fichero generado en la ejecución

Aplicación de instrumentación Para instrumentar las clases de acceso a ficheros se desarrolló la siguiente clase:

es.pfc.trazas.javassist.exec.all.EjecutarConTrazasInstrumentar

Esta clase carga la clase que se le pasa por parámetro, ejecuta su main e instrumenta las llamadas a los métodos de consulta de datos a ficheros de tipo DataInput. En este caso los métodos se instrumentan para obtener los datos del fichero de trazas en lugar de obtenerlos del fichero original, de forma que se puede comprobar si la ejecución ha sido correcta.

Aplicación de prueba En este caso se creó una aplicación compuesta por diversas clases con las que se realizan tanto accesos a la base de datos como a ficheros de datos almacenados en el filesystem del sistema. También recibe parámetros de entrada al main.

La clase de prueba generada realiza las siguientes acciones sobre el fichero de prueba:

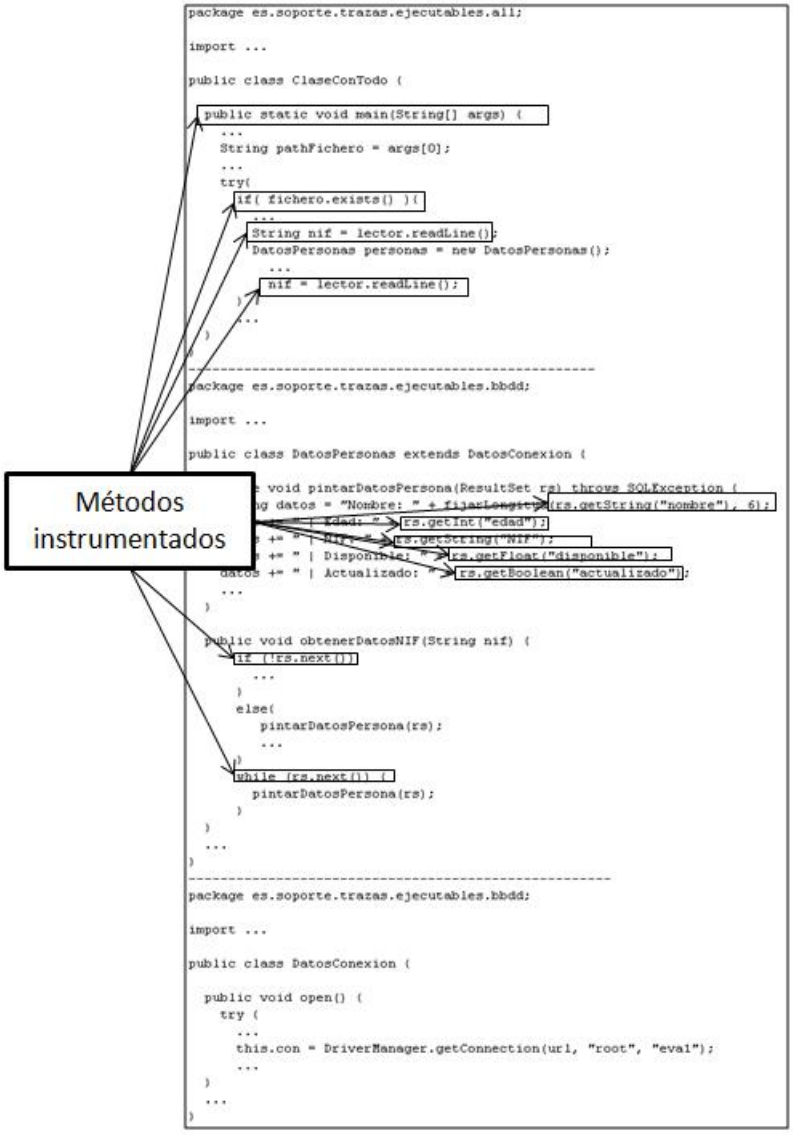


Figura 7.7: Clase de prueba para todos los accesos

1. Obtiene los datos de entrada de la aplicación. Espera un path a un fichero que existe.
2. Abre el fichero que se le pasa por parámetro y lo lee. El contenido del fichero es un conjunto de NIFs, cada uno en una línea del mismo.
3. Para cada NIF que lee en el fichero, consulta la tabla de personas de la bbdd, obtiene los datos y los pinta por pantalla.

La clase principal se llama *es.soporte.trazas.ejecutables.all.ClaseConTodo*.

La clase es la misma que en la ejecución, ya que intentamos comprobar la reejecución con los datos del fichero.

Resultado de la ejecución El resultado de la ejecución es el siguiente:

Leemos fichero: d:|pfc|pruebas|all|NIFs.TXT
———————————
NIF: 07721443M
Nombre: Eva | Edad: 30 | NIF: 07721443M | Disponible: 111.111 | Actualizado: true
———————————
———————————
NIF: 79271483M
No hay personas en la base de datos con NIF: 79271483M
———————————

Se puede comprobar que el resultado es el mismo que en la ejecución de la clase de forma independiente. Por tanto, la funcionalidad de la clase no ha cambiado. Esto es así porque en la instrumentación de la clase no modificamos el comportamiento, solo añadimos más (el almacenamiento de los datos en el fichero de trazas).

Trazas de la ejecución Las trazas de ejecución de la aplicación son las siguientes. En ellas se indican los métodos que se instrumentan:

Evita paquete:com.mysql.jdbc
Evita paquete:es.pfc.log
Instrumentamos método "main" de la clase ""
Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.all.ClaseConTodo"
Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "readLine" de la clase: "java.io.BufferedReader"

Instrumentado llamada al método: "readLine" de la clase: "java.io.BufferedReader"
Fichero: D:|Software|PFC|workspace|PFC|trazas.dat
El total de variables leídas: 14
Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.bbdd.DatosPersonas"

Instrumentado llamada al método: "next" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "next" de la clase: "java.sql.ResultSet"

Instrumentado llamada al método: "getString" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getInt" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getString" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getFloat" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getBoolean" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "next" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "next" de la clase: "java.sql.ResultSet"
Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.bbdd.DatosConexion"
Instrumentado llamada al método: "getConnection" de la clase: "java.sql.DriverManager"
Se indica en cada línea:

- 1 a 2 - En esta línea se indican los paquetes a evitar. Son dos: el del logger que nos sirve para leer los datos del fichero de trazas y el del fabricante de la base de datos. Si no los evitamos entramos en un bucle infinito.
- 2 - En esta línea se indica que se instrumenta la clase principal para que se cojan los parámetros de entrada del fichero de trazas.
- 3 - En esta línea se indica la primera clase que clase se instrumenta (*es.soporte.trazas.ejecutables.all.ClaseConTodo*), que en nuestro caso es la clase principal (donde esta el main) y por tanto es la primera que se carga. Las clases se van instrumentando mientras se van cargando.
- 4 a 6 - En estas líneas se indican que llamadas a métodos de la clase anterior se instrumenta.
- 7 - Path completo del fichero de trazas que se genera.
- 8- Indica el número de variables leídas del fichero de trazas. Son los datos con los que se ejecutará el proceso.
- 9 - En esta línea se indica la segunda clase que clase se instrumenta (*es.soporte.trazas.ejecutables.bbdd.DatosPersonas*).
- 10 a 18 - En estas líneas se indican que llamadas a métodos de la clase anterior se instrumenta.
- 19 - En esta línea se indica la tercera clase que clase se instrumenta (*es.soporte.trazas.ejecutables.bbdd.DatosConexion*).
- 20 - En estas líneas se indican que llamadas a métodos de la clase anterior se instrumenta. Se trata del establecimiento de la conexión a la base de datos, que falseamos con una existente en el sistema de reejecución para que no falle ahí y continúe la ejecución.

En la imagen 7.8 se puede ver que los métodos instrumentados son los correctos. Según las trazas se hacen las siguientes llamadas a los siguientes métodos:

- exists()

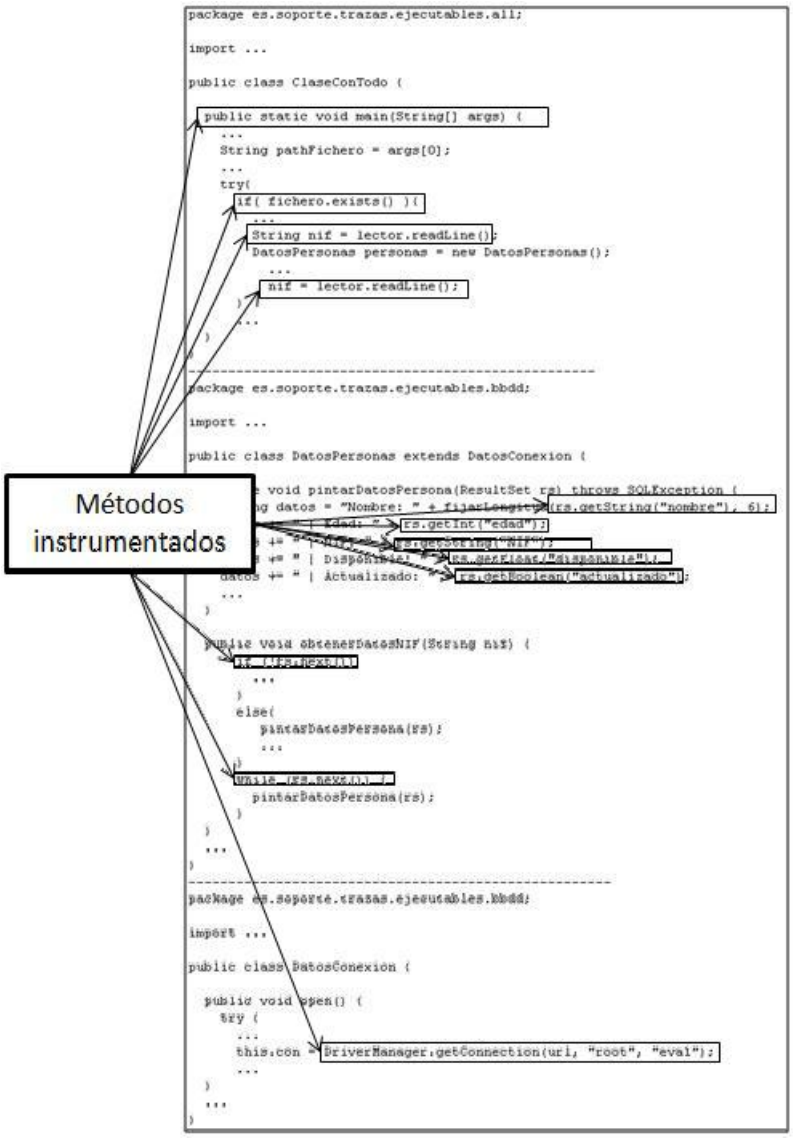


Figura 7.8: Clase de prueba para todos los accesos - Reejecución

- readBoolean()
- readByte()
- readChar()
- readDouble()
- readFloat()
- readInt()
- readLong()
- readShort()
- getConnection()

Estas llamadas son las que dicen las trazas que se instrumentan, por lo tanto la instrumentación parece correcta. Para comprobar que realmente se han instrumentado bien los métodos hay que mirar que los datos que se usan son los que se obtiene del fichero de trazas.

7.9.2.2. Reejecución a partir de un fichero creado/modificado manualmente

En este caso, para que la ejecución del proceso sea correcta, al modificar el proceso se ha de modificar la clase que lee. Esto es porque en la clase se lee el dato sabiendo de antemano el tipo de dato que se está leyendo. Modificamos el proceso y el fichero para obtener un dato más de tipo booleano.

Fichero de pruebas: Datos del fichero:
0 d:|pfc|pruebas|all|NIFs.TXT
1 true
2 07721443M
3 true
4 Eva
5 30
6 07721443M
7 111.111
8 false
9 false
10 null
A los datos que teníamos en el fichero eliminamos el último NIF consultado.

Resultado ejecución: El resultado de la ejecución del proceso es como solo hubiese un NIF en el fichero de NIFs a consultar en la bbdd.

Leemos fichero: d:\pfc\pruebas\all\NIFs.TXT

———————————

NIF: 07721443M Nombre: Eva | Edad: 30 | NIF: 07721443M | Disponible: 111.111 | Actualizado: false

———————————

Trazas de la ejecución Las trazas de ejecución son las mismas que en la reejecución, mas una línea más del dato de tipo booleano que hemos añadido al proceso de prueba.

Evita paquete:com.mysql.jdbc
Evita paquete:es.pfc.log
Instrumentamos método "main" de la clase ""
Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.all.ClaseConTodo"

Instrumentado llamada al método: "exists" de la clase: "java.io.File"
Instrumentado llamada al método: "readLine" de la clase: "java.io.BufferedReader"
Instrumentado llamada al método: "readLine" de la clase: "java.io.BufferedReader"
Fichero: D:\Software\PFC\workspace\PFC\trazas.dat
El total de variables leídas: 11
Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.bbdd.DatosPersonas"
Instrumentado llamada al método: "next" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "next" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getString" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getInt" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getString" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "getFloat" de la clase: "java.sql.ResultSet"

Instrumentado llamada al método: "getBoolean" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "next" de la clase: "java.sql.ResultSet"
Instrumentado llamada al método: "next" de la clase: "java.sql.ResultSet"
Instrumenta llamada a métodos de la clase: "es.soporte.trazas.ejecutables.bbdd.DatosConexion"
Instrumentado llamada al método: "getConnection" de la clase: "java.sql.DriverManager"

Como se puede ver la instrumentación es correcta. El número de variables leídas concuerda con las del fichero. Las llanadas a métodos instrumentados también concuerdan con las llamadas reales que se hacen.

7.10. Conclusiones

El objetivo de las pruebas ha sido el de probar los desarrollos realizados para comprobar la validez de los resultados obtenidos. El objetivo inicial del desarrollo ha sido el de encontrar una forma de almacenar los datos de entrada/salida de una aplicación.

Con los datos de test y las clases de prueba (simples) desarrolladas, se ha comprobado que se instrumentan de forma correcta las clases para obtener los datos de entrada/salida definidos en un principio (datos de entrada a la aplicación, acceso a bases de datos y acceso a datos almacenados en ficheros). En algunos de los casos incluso se ha conseguido más de una solución posible al problema inicial. En estos casos, el uso de uno u otro método se basaría únicamente en el performance conseguido por la aplicación en cada caso y en la forma en la que se lanzaría un proceso cualquiera (ya que en algunos casos se requiere de una clase intermedia que realice posteriormente la llamada a la aplicación que realmente queremos ejecutar).

Todo esto se ha probado bajo el supuesto de que las aplicaciones se ejecutan en un entorno controlado y sin errores (los cambios de flujo que producen estos en una aplicación no se han tenido en cuenta en esta fase de desarrollo, ya que no son datos de entrada/salida en si, pero si es necesario almacenar una marca de donde se cambia el flujo de la aplicación a causa de una excepción cualquiera). Estas suposiciones limitan bastante la aplicación de las soluciones encontradas, pero es una base sobre la que seguir desarrollando para encontrar una solución aplicable a aplicaciones reales.

En definitiva podemos decir que el objetivo del desarrollo (el almacenar los datos de entrada/salida) de una aplicación se ha conseguido de forma satisfactoria. Sin embargo, el conjunto de suposiciones fijadas (versión jre, ejecución sin errores) hace que la solución sea, aún, no aplicable en aplicaciones reales.

Bibliografía

[Vig98] G.Vigna. Cryptographic traces for mobile agents. In Mobile agents and Security, volume 1419 of LNCS. Springer-Verlag, 1998.

[DBLM098] Danny B.Lange and Mitsuru Oshima. Programming and Deploying Java Mobile Agents with Aglets. Adison-Wesley, 1998

[ASDK-Doc] Aglets Sofware Development Kit. Documentation. http://www.trl.ibm.com/aglets/

[ASDK-sdk] Aglets Sofware Development Kit. Download ASDK. http://sourceforge.net/projects/aglets/

[BCEL] http://jakarta.apache.org/bcel/

[Javassist] JAVa programming ASSISTant. Documentation and API manual. http://www.csg.is.titech.ac.jp/~chiba/javassist/

[Javassist SDK] JAVa programming ASSIST, sources and binaries. http://sourceforge.net/projects/jboss/files/Javassist/

[ChiMis03] Shigeru Chiba and Muga Mishizawa. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE '03), volume 2830 of LNCS. Springer-Verlag, 2003.

[TaSaChiIt01] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba1, and Kozo Itano. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. In ECOOP2001, volume 2072 of LNCS. Springer-Verlag, 2001.

[Aarniala05] Jari Aarniala. Instrumenting Java bytecode. In Seminar work for the Compilerscourse, spring 2005.

[ChaMiShin01] Ajay Chander, John C. Mitchell and Insik Shin. In DARPA Information Survivability Conference & Exposition (DISCEX). 2001.

[Recsi04] Oscar Esparza, Miguel Soriano.Detección y Prueba de Ataques en Sistemas de Agentes Móviles. In Reunión Española sobre Criptología y Seguridad de la Información (RECSI'04) , 2004. ISBN 84-7978-650-7 .

[JaKar800] Wayne Jansen, Tom Karygiannis. NIST Special Publication 800-19 – Mobile Agent Security

[Chiba98] S. Chiba. Javassist | A Reflection-based Programming Wizard for Java. In Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java, October 1998.

[Java] Source for Java Developers. Documentation and binaries. http://java.sun.com/

[JVM] JVM: Java Virtual Machine documentation. http://java.sun.com/j2se/1.5.0/docs/guide/vm/index.html

[MySQL] MySQL. Open Source database. http://www.mysql.com/.

[JVMTI] Java Virtual Machine Tool Interface. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/index.html

[Eclipse] Eclipse. Entorno de desarrollo integrado de código abierto. http://www.eclipse.org/

[LYX] Lyx, document processor. http://www.lyx.org/

[Argo] Argo. UML Modeling tool. http://argouml.tigris.org/