Università degli studi di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali

Dipartimento di Informatica

Tesi di laurea

EXaM: un tool per l'esecuzione e il monitoring di applicazioni a componenti distribuiti

15 dicembre 2006

Candidato:

Annalisa Bacherotti (`bacherot@cli.di.unipi.it`)

Relatori:
Ing. Antonia Bertolino
Dott. Andrea Polini

Controrelatore:
Prof. Carlo Montangero

Anno Accademico 2005/06

A mio padre.

EXaM: un tool per l'esecuzione e il monitoring di applicazioni a componenti distribuiti

Annalisa Bacherotti

Laurea in Informatica.

Sommario

Nella dissertazione viene presentato lo studio e lo sviluppo di un tool per la configurazione di un ambiente di monitoring per applicazioni a componenti software distribuiti. Viene inoltre illustrata una tecnica per il recupero delle informazioni rilevanti a livello architetturale, ossia tracce di esecuzione derivanti dall'osservazione delle interazioni tra i componenti a runtime.

Dichiarazioni

The work in this thesis is based on research carried out at the Institute of Information Science and Technologies (ISTI) of the Italian National Research Council (CNR) in Pisa, Italy. No part of this thesis has been submitted elsewhere for any other degree or qualification and it all my own work unless referenced to the contrary in the text.

Indice

Elenco delle figure

Parte I

L'architettura software e i sistemi di validazione

Capitolo 1

Introduzione

Con l'aumentare della complessità dei sistemi software e degli aspetti di interoperabilità che essi presentano, è emersa sempre più l'esigenza di meccanismi di verifica delle funzionalità offerte e della correttezza delle interazioni tra i componenti costituenti. Si assiste così a un rinnovato interesse per le tecniche di verifica in ambienti fortemente dinamici e difficili da analizzare, in cui la tempestività della scoperta di eventuali problemi di interoperabilità si rivela cruciale.

Congiuntamente al proporsi di questi argomenti in modo sempre più stringente, il linguaggio Java si è evoluto in una piattaforma matura per lo sviluppo di applicazioni d'impresa, facendo emergere anche la necessità di uno strumento che non solo favorisca la creazione di applicazioni formalmente corrette, eleganti e robuste, ma che faciliti anche la scrittura di codice efficiente e facilmente analizzabile. Questo scenario ha quindi portato alla richiesta di modalità innovative di analisi delle applicazioni Java nel momento stesso della loro esecuzione, soprattutto in sistemi distribuiti. Ciò che questo comporta è l'avvento di una nuova generazione di strumenti di debugging e di profiling, ma anche di tecniche di *monitoring* e di *testing* che permettano di effettuare operazioni di diagnosi e di fault-detection e che sopperiscano alle mancanze degli strumenti attuali, rivelatisi spesso inadeguati o dal costo eccessivo.

Con l'aiuto di una corretta formalizzazione delle proprietà che un sistema

deve rispettare e di tecniche che determinino se l'esecuzione corrente preservi tali proprietà, l'osservazione del comportamento di un programma in esecuzione, mirato a rivelare eventuali mancanze a livello di programmazione e di allocazione della memoria, costituisce uno strumento molto efficace al fine di portare alla luce stati anomali che possano indurre errori. La gestione di sistemi a componenti distribuiti su una rete secondo requisiti ben definiti, da cui dipende l'intera qualità del servizio, necessita di strumenti di validazione di questo tipo, che riescano a combinare le varie tecniche di monitoring e le metodologie di analisi dei risultati. E' a questo scopo che una corretta definizione progettuale dei requisiti secondo gli stili della *Software Architecture* si rivela fondamentale, poichè tale disciplina permette di organizzare un intero sistema, strutturato secondo i suoi componenti costituenti, in base alle relazioni tra di essi e con l'ambiente intorno, in virtù dei principi che devono governare la sua progettazione, la sua esecuzione e la sua evoluzione.

Il contributo che questa tesi cerca di offrire al panorama attuale di tecniche di testing e validazione è uno strumento per facilitare il monitoring architetturale di sistemi a componenti distributi. L'utilizzo di un tale strumento mira a facilitare l'integrazione della software architecture nell'intero processo di sviluppo, forzando gli sviluppatori a seguire tappe obbligate che costantemente verifichino l'adesione dei risultati, anche parziali o prodotti da un solo componente, ai requisiti, ai vincoli e alle proprietà che le analisi dei requisiti e progettuali hanno precedentemente sottolineato. Lo strumento, denominato EXaM (EXecution And Monitoring), utilizzando un'architettura centralizzata, consente di rilevare le interazioni tra componenti distribuiti, secondo le direttive impartite dall'utente per mezzo di un file di configurazione. La forza innovativa di EXaM consiste nella possibilità, lasciata all'utente, di decidere *a priori* i componenti di cui si vogliono ottenere le tracce di esecuzione, limitando così la produzione dei risultati a quelli ritenuti rilevanti a livello architetturale.

Il lavoro qui esposto è suddiviso in tre parti. Nella prima parte viene presentata l'Architettura Software e i sistemi basati sui componenti (capitolo 2), insieme a una breve rassegna delle tecniche di verifica e di testing e in particolare del monitoring a runtime (capitoli 3 e 4). Nella seconda parte vengono

esposte le motivazioni che hanno portato alla nascita di EXaM (capitolo 5) e le caratteristiche funzionali e tecniche di questo strumento (capitoli 6 e 7), soffermandoci in particolar modo sulla derivazione delle tracce di esecuzione per un semplice caso di studio (capitolo 8). La terza parte raccoglie i dettagli per effettuare il deployment di EXaM (in Appendice A) e alcuni estratti di codice attraverso una Application Programming Interface (in Appendice B).

Capitolo 2

Architettura software e sistemi basati sui componenti

Sebbene nel corso degli anni la Software Architecture possa talvolta essere stata considerata marginalmente, essa è destinata a svolgere una parte prioritaria nello sviluppo di qualsiasi sistema software che si proponga di essere affidabile, riusabile e facilmente migliorabile. Un'architettura software correttamente progettata al fine di soddisfare i necessari requisiti di qualità e diffusamente documentata è la chiave del successo di un'applicazione complessa.

Per Software Architecture (SA) si intende l'insieme delle principali decisioni di progettazione riguardo a un sistema, volte ad aumentare la qualità del software prodotto e a ridurre drasticamente sia i costi di sviluppo sia il tempo per immettere un prodotto sul mercato. Essa fornisce un'astrazione ad alto livello per la rappresentazione delle strutture, delle proprietà chiave e del comportamento di un sistema software complesso, definendo tutti gli elementi del sistema e le relazioni esistenti tra di essi.

La SA è una disciplina da integrare nella pianificazione dello sviluppo e nello sviluppo stesso; all'interno del ciclo di vita del software, essa non costituisce una fase preliminare ma si colloca piuttosto in tutte le fasi, apportando a ciascuna di esse un contributo significativo:

1. nella fase di analisi dei *requisiti***:** nel caso di modelli iterativi, ossia in quei modelli di sviluppo che permettono di rivedere e analizzare di nuovo le fasi precedenti, essa identifica le caratteristiche del sistema e le qualità che si desidera rispettare alla luce delle considerazioni fatte a livello progettuale. In particolare, se i requisiti vengono solo delineati in base alle richieste dell'utente, è possibile riesaminarli raffinandoli ulteriormente in seguito a dettagli emersi nelle fasi successive di progettazione e analisi;

2. nelle fasi di *progettazione* e di *analisi*: modella, aiuta a visualizzare e ad analizzare le decisioni progettuali corrispondenti ai requisiti. Tali decisioni possono essere guidate dall'adozione di uno o più *stili architetturali*;

3. nella fase di *implementazione* e di *testing*: essa guida il testing e rafforza l'utilizzo di politiche di sicurezza per prevenire eventuali errori che potrebbero verificarsi nel sistema;

4. nella fase di *manutenzione*: può essere utilizzata come base per aggiungere funzionalità senza violare i vincoli e le proprietà iniziali.

Ciò che fa un'architettura software è definire senza ambiguità i concetti che dovranno costituire il sistema software, come i componenti coinvolti, i sottosistemi, i livelli, le interfacce e le varie funzionalità. E' compito della SA, poi, scegliere la tecnologia da utilizzare, definire l'architettura del sistema in termini dei componenti definiti e organizzare di conseguenza il lavoro di sviluppo con l'aiuto di paradigmi e di strategie predefinite. Dopodichè, è necessario prevedere e affrontare eventuali problemi di performance, predisporre un ambiente di lavoro consono al progetto da sviluppare in termini di configurazioni hardware e software e infine documentare i componenti, le interfacce principali e la stessa architettura con la presentazione del problema, le possibili soluzioni, la soluzione adottata e le conseguenze di tale scelta.

Un'architettura software è dunque parte dello stesso sistema che deve rappresentare. Essa può essere analizzata, documentata e modificata durante

l'evoluzione del sistema rappresentato ed è necessaria per analizzare la consistenza dell'implementazione del sistema rispetto alla sua descrizione, cioè per verificare che le parti che la costituiscono cooperino in modo appropriato.

Per far sì che un'architettura riesca a rappresentare efficacemente i settori più disparati della progettazione e i diversi campi applicativi, è possibile utilizzare vari strumenti come i linguaggi basati sulla rappresentazione XML per la descrizione dei componenti e delle relazioni coinvolti, i tool di rappresentazione grafica delle entità in gioco o di analisi della consistenza di un'architettura e soprattutto insiemi di decisioni progettuali e di vincoli da applicare a determinati problemi per ottenere il livello di qualità desiderata, denominate *stili architetturali*.

E' possibile confrontare le diverse architetture basandoci su un'impostazione comune, che consiste nell'avere una serie di elementi computazionali (**componenti**) e le interazioni esistenti fra questi (**connettori**). Esempi di componenti sono i client, i server, i filtri, i livelli o i database; esempi di connettori sono le chiamate di procedure, il broadcast di eventi, i protocolli di comunicazione o i pipes.

2.1 Gli stili architetturali

Per **stile architetturale** si intende un insieme di regole, di vincoli progettuali e di decisioni indipendenti dall'implementazione che possono essere applicati a tutti quei sistemi che si prefiggano obiettivi specifici al fine di migliorare la qualità del prodotto risultante. L'applicazione di tali stili può avvenire in fase di modellazione, di costruzione o di analisi del sistema [21]. Poichè le decisioni progettuali basate sugli stili architetturali sono interamente documentate e motivate, è più facile per gli sviluppatori estendere il sistema senza minarne le basi fondamentali o violare i principi architetturali su cui esso si basa.

Uno stile architetturale definisce un'intera famiglia di sistemi in termini di paradigmi di organizzazioni strutturali, dal momento che fornisce un vocabolario progettuale di tipi di componenti, di connettori e di vincoli sulle modalità di combinazione. In alcuni casi esso riesce anche a fornire un mo-

dello semantico che specifichi come ottenere le proprietà dell'intero sistema dalle proprietà delle parti che lo compongono. E' compito quindi di uno stile architetturale definire, oltre ai componenti e ai connettori, quali siano i paradigmi strutturali ammessi, quale sia il modello computazionale alla base del sistema e quali le proprietà invarianti secondo tale stile.

La scelta di uno stile piuttosto che un altro è di solito guidata dal campo applicativo e da un'attenta valutazione di quali siano i vantaggi e gli svantaggi propri di ognuno. Secondo quanto esposto in [21], è possibile raggruppare i principali stili architetturali nei seguenti gruppi:

- **sistemi data flow**: vi appartengono gli stili di tipo *pipes and filters* e i sistemi *batch sequential*;
- **sistemi call-and-return**: ne fanno parte, ad esempio, i sistemi main program e subroutine, i sistemi object oriented e quelli a livelli gerarchici;
- **sistemi a componenti indipendenti**: in generale si tratta di sistemi a eventi, di sistemi a invocazione implicita e di sistemi costituiti da processi comunicanti tra loro (client/server, peer to peer);
- **virtual machine**: comprendono sistemi facenti uso di interpreti e sistemi basati sulle regole;
- **sistemi data-centered (o repository)**: appartengono a questo tipo i database, i blackboards e i sistemi ipertesto.

2.1.1 I sistemi data flow

Nei sistemi data flow il controllo della computazione è affidato alla disponibilità dei dati, che costituiscono peraltro l'unica forma di comunicazione tra i componenti. Il flusso ordinato dei dati da un componente all'altro determina la struttura di questi sistemi e le alterazioni che un sistema può subire dipendono strettamente dalla tipologia del grafo di connessioni, ovvero dal modo in cui i componenti sono distribuiti sulla rete e dal livello di concorrenza adottato.

In questo quadro i componenti sono costituiti dalle entità che si occupano di ricevere, elaborare e inoltrare il flusso dei dati: ciascun componente ha un'interfaccia realizzata per mezzo delle porte di input con cui leggere i dati in ingresso e delle porte di output con le quali scrivere i dati in uscita; i connettori sono invece costituiti dal flusso stesso dei dati unidirezionale e solitamente asincrono. Il modello computazionale di sistemi di questo tipo prevede la lettura dei dati in ingresso, l'elaborazione e la produzione dei dati in uscita.

2.1.1.1 Un esempio: Pipes and Filters

Questo stile richiede che ciascun componente *(filtro)* abbia due byte streams *(pipes)* dedicati alla comunicazione, uno per leggere i dati in ingresso e uno per produrre i dati in uscita. I pipes di ciascun componente funzionano così da connettori e quindi da veicoli delle informazioni e possono essere sia unidirezionali sia bidirezionali.

Pipes and Filters garantisce la compatibilità sintattica secondo la quale due filtri possono essere sempre connessi e comunicare. Inoltre, poichè lo stream di input viene trasformato localmente e in modo incrementale per permettere al flusso di uscita di iniziare prima che il flusso in ingresso sia stato elaborato totalmente, aumenta la parallelizzabilità delle applicazioni permettendo lo scambio dei risultati parziali. Questo stile, che prevede come modello computazionale un flusso di dati sequenziale, stabilisce come proprietà invarianti che i filtri siano entità indipendenti che non conoscono l'identità dei filtri alle estremità dei due pipes adiacenti, sebbene possano specificare il formato dei dati trasmessi. Le specializzazioni più comuni di questo stile includono *pipelines* che ammettono solo sequenze lineari di filtri o *bounded pipes* che limitano l'insieme dei dati gestibile da ciascun filtro.

Un esempio di architettura pipes and filters sono le shell di Unix, in cui i componenti sono rappresentati dai processi e le pipes sono create a livello di file system. Altri esempi ci giungono dai compilatori, dai sistemi di elaborazione dei segnali, dalla programmazione parallela, da quella funzionale e dai sistemi distribuiti.

Tra i principali vantaggi nell'utilizzo di Pipes and Filters abbiamo la semplicità del sistema ottenuto dalla composizione dei singoli componenti, la riusabilità, la facilità di manutenzione e di evoluzione dei filtri e dell'intero sistema, la possibilità di analisi specializzate e l'esecuzione concorrente. Tra gli svantaggi si annoverano invece la scarsa interattività tra i componenti e la difficoltà di sincronizzazione tra gli stream. Inoltre, l'assenza di informazioni comuni può indurre lavoro addizionale dal momento che ogni filtro deve effettuare il parsing dei dati in ingresso, influendo in questo modo sulle performance e sulla complessità di ciascun componente. Inoltre, poichè le trasformazioni dei dati avvengono senza stato, il campo applicativo risulta essere piuttosto limitato.

2.1.2 I sistemi call-and-return

I sistemi call-and-return prevedono la presenza di oggetti (i componenti), ciascuno con una serie di metodi pubblici, che si scambiano messaggi attraverso l'invocazione di metodi (i connettori). Solitamente un problema viene scomposto in una gerarchia di processi in cui il componente principale invoca i metodi pubblici degli altri componenti. Il loro aspetto più rilevante è la proprietà dell'incapsulazione, per mezzo della quale gli oggetti nascondono la loro implementazione e mantengono l'integrità dell'interfaccia. L'organizzazione in oggetti separati permette anche di suddividere i problemi in agenti indipendenti e interattivi che ben si adattano a scenari distribuiti.

2.1.2.1 Astrazione dei dati e organizzazione object-oriented

In linea con la programmazione object-oriented, questo stile prevede che i dati e le operazione a loro associate (i metodi) siano incapsulate in un tipo di dato astratto (l'oggetto): i componenti sono quindi rappresentati dagli oggetti e i connettori dalle chiamate dei metodi che operano su di loro. Gli oggetti, la cui implementazione è nascosta agli altri, vengono così ad essere gli unici responsabili della risorsa che devono gestire. Ciò costituisce uno dei principali vantaggi di questo stile, poichè è possibile cambiare l'implementazione di

un oggetto senza coinvolgere le altre entità coinvolte; inoltre, stimola gli sviluppatori ad avere routines indipendenti e agenti che interagiscono tra di loro.

A differenza di Pipes and Filters, però, questo stile prevede che ogni oggetto conosca l'identità degli altri e le funzionalità da loro offerte; inoltre introduce il problema dei side-effects quando due oggetti utilizzano contemporaneamente un terzo oggetto: se ad esempio A chiama il metodo p di B mentre C sta utilizzando lo stesso metodo, l'effetto della chiamata di C su B può avere effetti indesiderati anche su A.

2.1.2.2 Sistemi a livelli

Un sistema a livelli è organizzato gerarchicamente in modo che ogni livello fornisca un servizio al livello soprastante e fruisca di un altro servizio offerto da quello sottostante, proponendosi rispettivamente come server e come client. In alcuni sistemi, ciascun livello è nascosto a tutti gli altri tranne ai due adiacenti. Secondo questo stile, i connettori sono rappresentati dai protocolli che regolano le interazioni tra i livelli (i componenti) e possono essere sottoposti a vincoli con i livelli adiacenti. L'esempio più significativo è dato dal protocollo di comunicazione ISO-OSI (Open Systems Interconnection - International Standards Organization), in cui i livelli più bassi rappresentano le connessioni hardware e i livelli più alti le applicazioni a livello utente. La progettazione di questi sistemi si basa sull'aumento dei livelli di astrazione e permette di affrontare problemi complessi suddividendoli in una serie di passi più facilmente risolubili.

Se il vantaggio di un tale approccio consiste nella limitazione del numero dei livelli in cui si svolge una comunicazione e quindi nell'effettiva separazione dei compiti e nella ridotta necessità di apporre cambiamenti alla struttura quando non siano coinvolte le interfacce, esso però introduce anche vari problemi, dovuti soprattutto alla difficoltà di strutturare correttamente un sistema secondo tali regole. Può infatti risultare difficile stabilire il giusto livello di astrazione, ad esempio nel caso di sistemi reali la cui implementazione richiederebbe l'adozione di più di un livello per lo stesso componente.

Inoltre, i sistemi a livelli non favoriscono performance elevate nel caso in cui i livelli più alti debbano comunicare spesso con quelli più bassi.

2.1.3 I sistemi a componenti indipendenti

I sistemi a componenti indipendenti sono costituiti da un certo numero di oggetti indipendenti che comunicano attraverso lo scambio di messaggi. Data l'indipendenza di ciascun processo, l'aggiunta o la rimozione di un componente comporta un impatto minimo ai fini del mantenimento dell'integrità e della correttezza dell'intero sistema. Come esempio di questo stile architetturale è possibile citare la gestione del desktop di Gnome, composto da processi che svolgono i propri compiti specifici in modo assolutamente indipendente (ad esempio Gnome-clock) o coordinandosi con gli altri (come il menu proncipale di Gnome panel), comunicando per mezzo di file condivisi o chiamate di sistema.

2.1.3.1 Il sistema Event-based e lo stile di invocazione implicita

Questo stile prevede che ciascun componente, invece di invocare una procedura direttamente, annunci in broadcast uno o più eventi a cui gli altri oggetti, se interessati, rispondono con la registrazione dell'evento, cioè associandovi una procedura. Il sistema, poi, invoca tutti gli eventi registrati causando in questo modo l'invocazione implicita delle procedure associate. I componenti architetturali di questo stile sono moduli le cui interfacce forniscono un insieme di eventi e un insieme di procedure, le quali possono essere chiamate direttamente o possono essere registrate sugli eventi del sistema.

I sistemi a invocazione implicita sono utilizzati negli ambienti di programmazione che permettono di integrare tool esterni, nei sistemi di gestione di database che garantiscono i vincoli di consistenza e nelle interfacce utente che separano i dati dalla loro rappresentazione. Ritroviamo esempi di questo stile nei daemon o nelle reti packet-switched.

Il meccanismo dell'invocazione implicita facilita il riutilizzo dei componenti registrati su un certo evento e favorisce l'evoluzione del sistema permettendo ai componenti di essere sostituiti senza dover modificare le interfacce.

Nei sistemi con i repository condivisi di dati, però, la performance e l'accuratezza del gestore delle risorse può divenire critico. Inoltre, stabilire la correttezza del sistema può essere difficoltoso poichè il significato di una procedura che annuncia eventi dipende dal contesto in cui essa è invocata e infine non ci sono garanzie che i componenti che annunciano eventi ricevano una risposta.

2.1.4 I sistemi data-centered

I sistemi data-centered sono costituiti da un componente centrale che rappresenta lo stato dell'intero sistema e all'interno del quale i dati sono memorizzati e sottoposti alle elaborazioni degli altri componenti indipendenti e decentralizzati. Il caso più rappresentativo è dato dalle architetture blackboard, illustrate nel paragrafo seguente.

2.1.4.1 Repository e stile blackboard

Gli stili che adottano i repository sono costituiti da due diversi componenti: una struttura dati centrale che rappresenta lo stato corrente e un insieme di componenti indipendenti che operano sui dati immagazzinati nella struttura centrale. Se è la transazione in input a selezionare i processi su cui eseguire, allora è possibile utilizzare un database tradizionale come repository; se invece i processi sono selezionati dalla struttura centrale in cui sono memorizzati i dati, allora il repository segue un modello la cui configurazione prevede che più client condividano un solo *blackboard.* Questo modello ha solitamente tre componenti:

1. le *sorgenti di conoscenza*: le fonti di conoscenze specifiche di ogni applicazione sono rappresentate da componenti indipendenti, le cui interazioni avvengono solo attraverso la blackboard;

2. la *struttura dati blackboard:* un repository di dati condivisi e organizzati in una gerarchia indipendente dall'applicazione. Le sorgenti di cono-

scenza apportano modifiche allo stato, permettendo in questo modo l'avvicinamento alla soluzione del problema;

3. il *controllo* degli eventi: esso è guidato dallo stato della blackboard. Le sorgenti di conoscenza rispondono in modo opportuno ai cambiamenti nella blackboard.

In questo stile i componenti sono i programmi client che utilizzano la blackboard e l'unico connettore è la blackboard stessa. L'invocazione delle sorgenti di conoscenza dipende interamente dallo stato della blackboard, come prevede il modello computazionale alla base di questo stile, mentre il controllo degli eventi può essere implementato nelle sorgenti di conoscenza, nella blackboard o in entrambi. Proprietà stilistica invariante è la caratteristica che ogni client debba vedere tutte le transazioni nello stesso ordine.

Questo stile è solitamente utilizzato per applicazioni che richiedono un'interpretazione complessa dell'elaborazione di segnali. Sebbene si presti per tutte le applicazioni di rete, inclusi i database distribuiti, in presenza di un numero levato di client può indurre dei colli di bottiglia.

2.1.5 Virtual Machine

In realtà si tratta di un sottosistema delle architeture a livelli in cui un livello è realizzato come un interprete di un linguaggio e in cui i componenti sono il programma da eseguire e l'insieme dei dati su cui esso lavora.

2.1.5.1 Interpreti

Gli interpreti, la cui funzione principale è quella di creare una Virtual Machine, sono solitamente costituiti da quattro componenti: il programma da interpretare, lo stato del programma, lo stato dell'interprete e l'interprete stesso. Questi componenti vengono utilizzati per assottigliare la distanza tra il motore di calcolo a livello hardware e la semantica di un programma. Teoricamente, i linguaggi di programmazione come il Pascal, il Basic e Java costituiscono una virtual *language* machine.

2.1.6 Altri sistemi

Oltre a quelle viste, esistono molte altre implementazioni per lo stesso tipo di sistemi *multiprocess*. Alcuni sono definiti dalla topologia (ad esempio a stella o ad anello), altri sono caratterizzati dal tipo di protocolli adottati per le comunicazioni tra i processi.

2.1.6.1 Client - server

Un'architettura tipica comune a molti sistemi distribuiti è l'architettura *client-server* in cui un componente (il server) fornisce servizi agli altri (i client) senza conoscere il numero né l'identità dei client. Sono infatti i client a conoscere l'identità del server o a trovarla attraverso un altro server (name-server) con questa specifica funzionalità, e ad accedere ai servizi attraverso chiamate remote (*Remote Procedure Call*).

2.1.7 Architetture eterogenee

Spesso i sistemi adottano non un solo stile ma una combinazione di stili differenti. Ad esempio, i componenti di una struttura gerarchica possono avere una struttura interna sviluppata secondo un metodo diverso o utilizzare ciascuno connettori differenti, mentre i connettori possono essere scomposti secondo altri sistemi (ad esempio i pipes possono essere implementati come code FIFO). Un esempio di questa architettura sono i sistemi Unix pipes and filters in cui il file system agisce da repository, riceve il controllo da switch di inizializzazione e interagisce con gli altri componenti attraverso pipes:

```
ls -l ~ | grep -v ''d'' | sort -nr | mail -s ''Directory listing''
mymail@gmail.com
```

2.2 Il paradigma Component-Based

La maggior parte delle metodologie per l'analisi e il testing basati sulla Sofware Architecture assumono un approccio guidato dal modello, in cui cioè le specifiche SA costituiscono il modello di riferimento e in cui il sistema

è soggetto a una validazione accurata e totale delle proprietà architetturali richieste prima del deployment, ossia prima che entri in produzione.

I passi avanti fatti nella Software Architecture hanno largamente contribuito all'avvento del paradigma di sviluppo basato sui componenti, proprio in virtù del fatto che le specifiche SA forniscono delle linee guida per lo sviluppo di un sistema ottenuto dalla composizione *corretta* - validata cioè di volta in volta - del pezzi di software al fine di arrivare al sistema nella sua totalità [18].

Per **componente** si intende un'unità software riusabile e indipendente, utilizzabile singolarmente o in combinazione con altri e con specifiche funzionalità definite sintatticamente da interfacce pubbliche, al fine di gestire un evento specifico o un insieme di eventi. Ogni componente nasconde all'esterno la propria implementazione interna in modo da poterla cambiare senza interferire con le interazioni con gli altri e in modo da poter costruire sistemi in cui i singoli componenti possano essere sostituiti con altri mantenendo invariato il funzionamento globale.

Possiamo dividere i componenti in tre categorie:

1. i dati o elementi architetturali (componenti), cioè elementi che contengono informazioni da acquisire, elaborare o trasformare;

2. i processori o elementi di elaborazione, cioè elementi a cui spetta il compito di trasformare i dati;

3. i connettori o elementi di connessione, quegli elementi che legano insieme i pezzi che compongono l'architettura e permettono l'assemblaggio dei componenti e le interazioni tra di essi. Essi possono essere definiti anche come *glue-code*, poichè costituiscono il collante (glue) che lega i vari componenti. Le chiamate alle procedure, i dati condivisi e i messaggi tra i componenti sono esempi di elementi di connessione (connettori) che legano gli elementi architetturali.

Il ruolo di ciascun componente e dei connettori[1] deve essere esaurientemente descritto a livello dell'architettura software, senza scendere però nei dettagli implementativi.

Un **sistema basato sui componenti** è dunque ottenuto dall'assemblaggio di tutti quei componenti che svolgono le funzionalità desiderate, secondo le regole e i passi dettati dall'architettura in fase di progettazione. Un tale sistema è facilmente migliorabile e manutenibile, poichè per apportare modifiche è sufficiente sostituire il componente che presenta degli errori o che appare datato, e permette inoltre di ridurre i tempi di sviluppo con ulteriori vantaggi sull'affidabilità e sulla qualità del prodotto offerto. Del resto, i sistemi software più complessi richiedono necessariamente un meccanismo di decomposizione perché siano più facilmente trattabili e perché si possa più facilmente ragionare sulle proprietà dell'insieme attraverso la comprensione delle proprietà dei singoli componenti.

Alcuni linguaggi come i Module Interconnection Languages (MILs) e gli Interface Definition Languages (IDLs) hanno già in passato fornito delle notazioni per descrivere sia le unità composizionali attraverso interfacce ben definite, sia i meccanismi per legare le singole unità. Secondo il modello previsto da questi linguaggi, ciascun modulo fornisce un insieme di funzionalità disponibili all'esterno e ne richiede altre fornite dagli altri moduli. Lo scopo del glue-code è quello di risolvere le relazioni di offerta/richiesta indicando, per ciascuna funzionalità, dove sia possibile trovare la sua definizione [17].

Definire un'architettura software per tali sistemi significa non tanto occuparsi del modo in cui un componente effettua una certa computazione ma del modo in cui tale computazione si combina con le altre nel sistema intero. Infatti, se prima era necessario ragionare in modo gerarchico, per cui la correttezza di un modulo dipendeva dalla correttezza dei moduli componenti, adesso essa è indipendente da quella dei moduli con cui interagisce e di cui è costituito. In questa nuova ottica, è la correttezza dell'intero sistema che dipende dalla quella dei singoli sottosistemi e delle loro interazioni [17].

Nell'ottica dei sistemi basati sui componenti, la SA viene ad essere il mo-

[1]Nel resto di questo capitolo si parlerà di *componente* nell'accezione indicata dai punti 1 e 2 e di *connettore* nell'accezione specificata nel punto 3.

dello che porta all'assemblaggio di un insieme di componenti per formare un sistema che soddisfi alcuni requisiti. Se nello sviluppo di tali componenti lo scopo è quello di creare entità riusabili con interfacce ben definite e la cui qualità e il cui funzionamento siano comprovati, durante lo sviluppo di un sistema a componenti (CBS), lo scopo viene ad essere il loro assemblaggio al fine di costruire un sistema di qualità. Durante la definizione di una Software Architecture di un CBS (CBSA), invece, l'obiettivo è quello di fornire linee guida ad alto livello sui componenti e i connettori che lo costituiscono, sul modo in cui i componenti debbano essere assemblati e sul modo in cui debbano avvenire le interazioni tra di essi - secondo gli stili, i vincoli e le regole specificati - giocando un ruolo fondamentale nella validazione della qualità del sistema assemblato.

Capitolo 3

Tecniche di verifica

Esistono svariate tecniche di verifica della qualità di un software, della sua aderenza ai requisiti richiesti dall'utente e della compatibilità con le specifiche iniziali. Di seguito sono elencate le principali tecniche che rappresentano diversi approcci per procedere con la verifica, ciascuno con modalità e scopi differenti: il model checking, il testing e il profiling. Della necessità e della validità delle tecnologie di monitoring si discuterà diffusamente nei due capitoli successivi.

3.1 Model checking

La tecnica del model checking permette di definire un modello del sistema con una notazione formale, specificare le proprietà richieste espresse secondo formule logico-temporali[1] e verificare in modo automatico se il modello soddisfa le specifiche formali [25].

La necessità di verificare la correttezza di un modello deriva dall'inevitabilità della generazione di errori nello sviluppo di un sistema a fronte della presenza di errori nel modello che rappresenta l'insieme dei requisiti

[1]La logica temporale descrive un sistema di regole per la rappresentazione di proposizioni qualificate in termini di tempo ed è largamente utilizzata per le verifiche formali, soprattutto per descrivere i requisiti di un sistema hardware o software. Ad esempio, un'espressione del tipo "*ogni volta che* viene fatta una richiesta, l'accesso alla risorsa viene *alla fine* garantito, ma non è *mai* assegnato a due richiedenti contemporaneamente" viene facilmente resa con la logica temporale.

progettuali e di sistema. Gli errori indotti da cattivi requisiti possono essere evitati con un modello che soddisfi ampiamente le richieste dell'utente, che siano comprensibili, privi di ambiguità, flessibili e modulari in previsione di cambiamenti futuri ed espressi secondo un linguaggio formale e ad alto livello, senza dettagli tecnici o implementativi. A tal scopo, il model checking si è imposto come principale tecnologia per la verifica dei requisiti per molti sistemi real-time embedded e critici da un punto di vista della sicurezza.

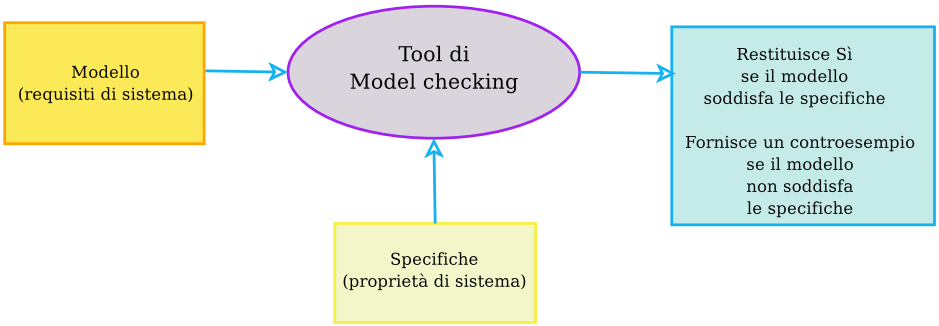


Figura 3.1: L'approccio nel model checking

In Figura 3.1 è illustrata l'idea alla base del model checking. Un tool di model checking accetta i requisiti di sistema (il *modello*) e una proprietà (la *specifica*) che il sistema finale deve soddisfare [26]. Il risultato dell'applicazione del tool agli input è positivo se il modello in esame soddisfa la data specifica ed è negativo nel caso contrario, generando inoltre un controesempio che fornisce un caso in cui la proprietà non è rispettata. Il controesempio fornito permette di individuare cosa ha causato l'errore, correggere il modello e sottoporre a test il nuovo modello. In questo modo è possibile avere un modello che soddisfi un numero sufficiente di proprietà di sistema affinchè si possa definire corretto.

I tool di model checking necessitano anche di un linguaggio formale che sia sufficientemente espressivo. Per i sistemi orientati al controllo si utilizza solitamente un'estensione degli automi a stati finiti (Finite State Machines) chiamata ESFM (Extended Finite State Machines) o una sua variante, che, per essere adeguata alla modellizzazione dei sistemi complessi, introduce anche la possibilità di differenziare i requisiti secondo diversi livelli di dettaglio

e di combinarli in rapporto ai componenti. Come esempio è possibile citare il symbolic model verifier (SMV) proposto da Carnegie-Mellon University come tool sufficientemente espressivo e dettagliato per verificare le proprietà di un sistema. SVM, come altri tool che seguono lo stesso approccio, rappresenta l'insieme degli stati in cui un sistema può trovarsi come un *albero di esecuzione* (o computazione), in cui la radice è etichettata con lo stato iniziale e ciascun figlio con lo stato successivo e su cui è possibile individuare un numero potenzialmente infinito di percorsi, detti *esecuzioni di sistema.* Le proprietà dei percorsi e degli stati che li compongono sono definite con la Computation Tree Logic (CTL), una notazione semplice e intuitiva, derivante dalla logica temporale ed estensione della logica proposizionale booleana (possiede infatti i connettivi logici *and, or, not, implies,* insieme ai connettivi temporali). Secondo quest'ottica, lo scopo del model checking consiste nell'esaminare se un dato albero di esecuzione soddisfi la specifica di una certa proprietà voluta dall'utente finale del sistema.

Tra i framework più interessanti nella progettazione dello sviluppo del software e nella sua validazione a fronte dei requisiti funzionali citiamo **CHARMY (CHecking ARchitectural Models consistencY)** [23, 1], che si avvale di tecniche varie di model checking tra cui le notazioni a stati finiti e lo *SPIN Model Checker* [22], e **Fujaba** [6], un tool che combina i diagrammi di classe UML con quelli comportamentali per ottenere un linguaggio di specifica potente, semplice e sufficientemente formale. Citiamo infine il linguaggio architetturale **Wright [16]** che fornisce una base formale precisa e astratta per la descrizione architetturale di sistemi software più o meno complessi, puntando soprattutto sui tipi di connettori espliciti, sull'utilizzo del controllo automatico delle proprietà architetturali e sulla formalizzazione degli stili architetturali. Wright inoltre fornisce un insieme di controlli sulla completezza e sulla consistenza da utilizzare in fase di progettazione del sistema secondo le tecniche del model checking.

I software che effettuano model checking offrono un approccio algoritmico esaustivo ed automatico per un'analisi completa del sistema, ciascuno con il proprio linguaggio di modellizzazione. Poichè però il numero di stati presenti in un modello può essere molto alto nonostante le tecniche di riduzione adot-

tate dai software, spesso il lavoro dell'utente di un tool di model checking può rivelarsi lungo e oberoso, poichè consiste nella verifica di ogni sequenza di stati in un percorso o nella riduzione dello spazio degli stati attraverso la riduzione dei dominii delle variabili. Ciò nonostante il model checking è senza dubbio un'ottima tecnica per verificare i requisiti progettuali di un dato sistema e scoprire errori nelle specifiche, permettendo dunque un notevole risparmio di tempo e di lavoro nelle fasi successive dello sviluppo di un sistema.

3.2 Software Testing

Nello sviluppo di un software, il testing è la fase mirata a stabilire la completezza, la correttezza, la sicurezza e la qualità del prodotto sviluppato. Uno dei suoi scopi principali è quello di eseguire l'applicazione in oggetto al fine di trovare eventuali errori e di raccogliere dati e informazioni sull'esecuzione da confrontare con le specifiche.

Esistono molti approcci al software testing, diversi secondo i componenti da testare, il contesto, l'ambiente in cui avvengono i test e ovviamente l'applicativo da sottoporre a test. Uno dei problemi più frequenti della fase di testing è dato dall'enorme presenza di possibili errori che si possono verificare a seconda delle molteplici e potenzialmente infinite configurazioni che si possono verificare: i *bugs* che avvengono raramente e solo in presenza di configurazioni specifiche e poco diffuse sono ovviamente difficilmente riproducibili in fase di testing. Da qui deriva la necessità di formalizzare il processo di software testing e di evidenziare una scala di priorità e di test da effettuare affinchè la maggior parte dei problemi venga riprodotta prima del deployment. E' necessario anche sottolineare la limitazione intrinseca di qualsiasi tipo di testing, ossia che esso riesce solo a provare la presenza di errori e non a dimostrarne la totale assenza.

Nei seguenti due paragrafi sono presentati altrettanti tipi di software testing tra i più diffusi e dall'ampio spettro di applicabilità. Nel successivo invece sono descritte le fasi sequenziali di testing a cui un sistema complesso

dovrebbe venir sottoposto: lo unit testing, l'integration testing, il system testing e l'acceptance testing

3.2.1 Black box e white box testing

Per testing black-box e white box si intende rispettivamente l'approccio dall'esterno e quello dall'interno di un singolo componente o di un sistema intero.

L'approccio black box si adotta quando non si conosce l'implementazione del componente e lo si può solo osservare da una prospettiva esterna, selezionando input validi e non validi e determinando gli output corretti da validare poi con gli output effettivi derivati dall'esecuzione del componente. Un tale approccio, oltre che per i software di cui non si conosce l'implementazione, è anche utilizzato per i livelli alti dei sistemi complessi, in cui è necessario semplificare e testare le funzionalità concettuali senza scendere nei dettagli tecnici e implementativi.

Il testing white box, detto anche testing strutturale, parte dalla conoscenza dell'implementazione interna di un sistema o di un componente per progettare casi di test basati sulla struttura interna. Chi sceglie i casi di test con questo approccio, oltre ad avere una approfondita conoscenza del codice, deve anche prevedere tutti i percorsi e tutte le combinazioni possibili. Il white box testing è solitamente utilizzato per testare le singole unità o le interazioni tra le unità al momento dell'integrazione dei componenti.

3.2.2 Automated testing

Il testing automatico è la verifica di un programma da parte di un computer. L'elaboratore che effettua il testing deve eseguire in automatico la procedura che prevede la lettura di alcuni casi di test, l'esecuzione del programma sotto esame secondo i test case in input e il confronto dei risultati prodotti con quelli attesi. Un tale test si rileva efficace in caso di programmi che eseguono azioni in conseguenza a dati che arrivano dalla rete o letti da supporti vari, mentre è poco adatto quando il software da testare richiede l'intervento di un utente o quando si tratta di testare un'interfaccia grafica, sebbene esistano

tool di scripting capaci di controllare ed eseguire test su applicazioni GUI. Esistono molte classi di problemi per cui i test automatici sono utilizzati sistematicamente; nella maggioranza dei casi, i test case sono generati secondo la tecnica del *model-based testing* dove, per generarli, si utilizza un modello del sistema.

3.2.3 Il software testing di un sistema complesso

Per i software complessi si adotta di solito una metodologia di testing più ampia, che va dal testing della singola unità (un modulo, un componente o una classe) fino al testing del sistema finito, passando per varie tipologie di verifiche e cercando di coprire il maggior numero di casi che possano verificarsi una volta che il software arrivi in produzione. Di seguito vengono descritti i principali passi delle verifiche a cui i sistemi vengono di solito sottoposti.

3.2.3.1 Unit testing

Lo unit testing è la procedura da eseguire su un modulo specifico per verificarne la correttezza. Tale procedura, utilizzata solitamente dagli stessi sviluppatori, consiste nello scrivere casi di test per ciascuna funzione presente nel modulo, in modo da poter identificare rapidamente quali elementi abbiano generato errori, anche a seguito di cambiamenti nel codice. Dal momento che ciascun test case viene condotto separatamente dagli altri, lo unit testing riesce a isolare le singole parti di codice e a localizzare eventuali malfunzionamenti, facilitando l'integrazione dei moduli la cui correttezza è stata dimostrata in un procedimento *bottom-up* - dal testing delle unità atomiche al testing dell'intero sistema ottenuto con la composizione delle unità. Ovviamente, un test di questo tipo non riesce a stabilire la correttezza del codice né al momento dell'integrazione, né tantomeno ad integrazione avvenuta. Esso è effettivamente funzionale solo se utilizzato congiuntamente ad altre attività di software testing, come il testing d'integrazione e quello di sistema, descritti nei paragrafi successivi.

3.2.3.2 Integration testing

L'integration testing (detto anche *I&T* da Integration & testing) è la fase che, nell'ottica di un'attività sistematica di software testing, segue lo unit testing e precede il system testing e consiste nell'integrare i singoli moduli software per poi verificare l'unità risultante. L'integration testing prende in input i moduli precedentemente testati singolarmente con lo unit testing, li ragguppa secondo criteri specifici ed esegue su ciascun gruppo così costituito i test descritti nel piano di test da seguire: l'output che ne deriva è il sistema integrato, il quale a sua volta dovrà sottoporsi al testing di sistema. Il testing I&T ha lo scopo di verificare i requisiti funzionali, di affidabilità e di performance definiti in fase di progettazione e di scoprire eventuali inconsistenze tra le unità software raggruppate e integrate o tra i gruppi stessi di unità. I casi di test vengono delineati affinchè si verifichi se tutti i componenti interagiscono correttamente, utilizzando, ad esempio, chamate di procedure o attivazione di processi, costituendo così un approccio di tipo *building block*, in cui i moduli già raggruppati (secondo tecniche di tipo *bottom-up*, *top-down* o altre ancora) e già verificati costituiscono una base che può essere dichiarata funzionante e che contribuisce a supportare il testing d'integrazione dei rimanenti moduli in attesa di verifica.

3.2.3.3 System testing

Questo tipo di testing viene condotto su un sistema integrato e ormai completato per verificare se esso sia conforme alle specifiche iniziali. Poichè non è richiesta alcuna conoscenza del codice o della logica interna al sistema, un test di questo tipo rientra nella classe dei black box. Il sistem testing prende in input tutti i componenti che sono stati integrati insieme e che hanno superato il test d'integrazione, insieme al sistema software stesso, per cercare di scoprire difetti sia a livello di composizione dei componenti sia a livello dell'intero sistema. L'approccio del system testing può dirsi di tipo distruttivo, cioè quello di assumere la presenza di errori rispetto alle specifiche funzionali e di sistema e di doverli dunque trovare sia nella progettazione sia nel comportamento in esecuzione, cercando di supporre le azioni e le aspettative di

un utente. I tipi di test che vengono condotti possono essere di vario tipo: funzionale, di usabilità, a livello di interfaccia, di performance, di affidabilità e molti altri. Questa è la fase immediatamente precedente a quella finale, il test di accettazione.

3.2.3.4 Acceptance testing

Il test d'accettazione, costituito da una serie di test detti *casi*, viene effettuato sul sistema che ha già superato le fasi precedenti di testing. Ciascun caso consiste nel verificare il comportamento del sistema se sottoposto a una certa condizione, derivante dall'ambiente o dal verificarsi di determinate proprietà: il risultato prodotto sarà un valore che indica il successo o il fallimento del test.

Solitamente quest'ultima fase del testing viene effettuata da utenti che agiscono secondo le condizioni dettate dal blck-box testing, ossia senza conoscere nessun dettaglio sul funzionamento interno del sistema: a fronte degli input precedentemente forniti e selezionati, si confrontano i risultati ottenuti con quelli attesi e se essi coincidono per ciascun caso, il test può dirsi superato.

Il test d'accettazione ha lo scopo di provare che il sistema finale incontra le richieste dell'utente concordate nella fase iniziale ed è utile anche per scoprire eventuali mancanze che non siano emerse nelle fasi precedenti.

3.3 Monitoring

Data la complessità degli attuali sistemi software, spesso coinvolti in più di un dominio applicativo, le tecniche di verifica e di validazione si possono rivelare insufficienti a indagare su eventuali difetti nel codice sviluppato. Se infatti i possibili comportamenti a runtime sono molteplici e spesso impredicibili, i vincoli temporali a cui sempre più i progetti sono sottoposti rendono impossibile un testing sufficientemente accurato da coprire interamente tutti i casi possibili. E' dunque necessario combinare queste tecniche con strumenti di monitoring per verificare se il software si comporti secondo le aspettative.

In realtà, si tende ad abusare del termine monitoring, spesso identificando una qualsiasi analisi a runtime. Una descrizione più precisa propone di classificare tale analisi in tre fasi ben specifiche:

1. la raccolta delle informazioni;
2. l'analisi rispetto a un comportamento o a un modello atteso (spesso fornito dal model checking);
3. la reazione in caso di divergenze.

Di solito, con *runtime software-fault monitoring,* o più semplicemente monitoring, si intendono i primi due punti descritti sopra, ossia l'attività di raccolta delle informazioni riguardo il comportamento di un sistema al fine di verificare se alcune proprietà specifiche vengano preservate durante l'esecuzione corrente. Il terzo punto è spesso delegato a un componente esterno con il compito di gestire il verificarsi di eventuali errori.

Il monitoring è comunque una fase obbligatoria per evitare errori o comportamenti anomali nell'esecuzione di un programma e può anche supportare l'attività di testing con dati e casistiche addizionali. E' stato inoltre spesso utilizzato per il profiling, per le analisi delle performance, per l'ottimizzazione del software, per scoprire eventuali difetti nel codice (*software fault detection*) e anche per le diagnosi e 'recovery' di errori.

3.3.1 Profiling

Alcuni aspetti problematici di un'applicazione non emergono né al momento del debugging né durante la fase di testing, ma solo in ambiente di produzione, quando il sistema è in esecuzione continua e sottoposto a grosse quantità di dati e ad alti picchi produttivi. E' a causa di questo che, prima del deployment, è necessario utilizzare uno strumento di profiling che ne analizzi l'esecuzione e identifichi eventuali problemi di performance.

Il profiling è un'attività di monitoring mirata a individuare eventuali problemi di *poor programming*, quali algoritmi inefficienti, memory leaks o bottlenecks, che possano indurre casi di *poor performance* [27]. Attraverso le

tecniche di profiling, è possibile evidenziare blocchi a livello I/O, creazione eccessiva di oggetti e uso inefficiente della memoria così come l'allocazione della CPU, il tempo di esecuzione di un certo metodo e quante volte esso venga invocato. Per linguaggi ad alto livello come Java in cui la presenza di una Virtual Machine può nascondere le implicazioni ai livelli sottostanti, piuttosto che per altri strumenti di programmazione, analizzare a runtime tali meccanismi è divenuto ormai critico, soprattutto con l'avanzare delle piattaforme verso ambienti totalmente distribuiti ed eterogenei e con il conseguente aumento della mole di dati su cui lavorare, al fine di migliorare le performance e il consumo di risorse critiche come la CPU e la memoria.

Solitamente, uno strumento di profiling è costituito dai seguenti tre componenti:

1. **Hook**: la parte che effettua il monitoraggio a runtime dell'applicazione e genera una serie di eventi al verificarsi di alcune condizione critiche;
2. **Agent/Handler**: la parte predisposta a ricevere gli eventi segnalati dall'hook, ad elaborarli e a memorizzare i dati generati su un opportuno repository per una successiva consultazione;
3. **Viewer**: il visualizzatore dei dati ricavati dall'agent, opportunamente selezionati e raffigurati in forma grafica o tabellare. Data la mole dei dati ricavati precedentemente, la fase di selezione e i criteri per effettuarla si rivelano critici e degni di particolare attenzione.

Un tool di profiling per applicazioni Java può operare secondo vari meccanismi:

- secondo la tecnica di instrumentazione del codice, o *wrapping*, ciascun metodo da monitorare viene sostituito con un altro con gli stessi parametri, da chiamare in sostituzione al metodo originario e che scriva in modo appropriato i dati necessari al profiling, fornendo informazioni

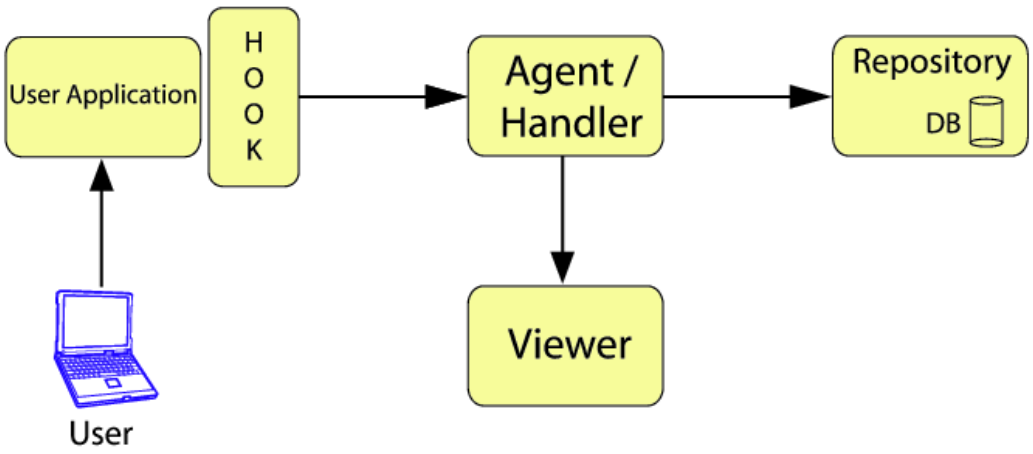


Figura 3.2: Lo schema di un profiler

sullo stato di terminazione della funzione. E' possibile anche instrumentare solo il bytecode (*BCI - bytecode instrumentation*) lasciando così il codice sorgente inalterato e operando solo sui file `.class` o anche sugli archivi `.jar`. Tale tecnica, meno invasiva della precedente, opera sul ClassLoader sostituendo al momento opportuno il `.class` del file da monitorare con un altro file `.class` da passare alla JVM che generi delle tracce utili al profiling. L'utilizzo dei tool di instrumentazione del codice, soprattutto del codice sorgente, è sconsigliato poichè comporta una modifica del codice da monitorare e introduce una dipendenza esplicita dallo specifico strumento di profiling.

- un altro strumento di profiling è disponibile con la libreria JVMTI, introdotta da SUN con la jdk 1.2 (con il nome di JVMPI - Java Virtual Machine Profiler Interface - e poi rinominata JVMTI - Java Virtual Machine Tool Interface), di cui si parlerà diffusamente nel capitolo 9. JVMTI fornisce un protocollo di comunicazione tra la JVM e un agent, ossia un modulo che seleziona ed eventualmente elabora gli eventi generati a livello della JVM: ciò che avviene a livello di thread viene osservato e, nel caso si verifichi un evento atteso o il thread interessato rientri tra quelli da osservare, lo stack di chiamate con le informazioni relative alle risorse hardware viene fornito al profiler. Uno strumento

di questo tipo, però, dovendo accedere allo stack di chiamate, altera l'ambiente di esecuzione provocando rallentamenti nelle chiamate dei metodi.

Esistono molti strumenti di profiling disponibili. **JRat** permette di instrumentare sia il codice sorgente sia il codice compilato, memorizza le tracce rilevanti in un file `.xrat` e le rende disponibili per la visualizzazione attraverso JRat Desktop. Analogamente, **prof** e **gprof,** disponibili per molte piattaforme Unix, instrumentano il codice eseguibile per produrre dati utili al profiling. **Eclipse Profiler,** il tool di profiling fornito da Eclipse, utilizza invece la libreria JVMTI. Un altro tool fortemente integrato con Eclipse è Eclipse Test & Performance Tools Platform (**TPTP**), utilizzato per il profiling dei plug-ins di Eclipse e di molte applicazioni Java, dalle semplici applicazioni stand-alone a quelle più complesse in esecuzione su piattaforme differenti. TPTP permette di analizzare i dati su cui opera l'applicazione, di identificare i metodi con i tempi di esecuzione più alti, di passare al codice sorgente durante l'esecuzione e di risolvere eventuali problemi di performance.

Citando [27], poichè "Java consente allo sviluppatore di scrivere applicazioni senza porre le dovute attenzioni al consumo di risorse e alle performance che queste avranno a runtime", è opportuno utilizzare uno strumento di profiling "non solo *a posteriori* rispetto allo sviluppo, ma come una pratica sistematicamente integrata nel processo di sviluppo".

3.3.2 Monitoring funzionale

In generale, il monitoring mira a trovare prove che il software in questione si comporti o meno secondo le proprietà specificate durante la sua esecuzione. Al contrario di tecniche come il testing, il theorem proving[2] e il model checking, il cui scopo è quello di provare la correttezza *universale* dei programmi, lo scopo del monitoring è quello di determinare se l'esecuzione *corrente* sia

[2]Nel theorem proving si esprime il sistema come un insieme di assiomi e regole d'inferenza su cui dimostrare, con tecniche di logica temporale, le proprietà viste come teoremi. A differenza del model checking, non restituisce alcun controesempio in caso di errore ed è difficilmente automatizzabile, sebbene esistano tool che effettuano il theorem proving.

corretta, ossia se essa conservi le proprietà specificate a livello architetturale, come descritto in [20].

I casi di esecuzione anomala che il monitoring deve rivelare per impedire che si verifichino dopo il deployment sono i seguenti:

- fallimento software (*software failure*): con questo termine si intende una divergenza tra il comportamento *osservato* del sistema software e il comportamento *richiesto;*
- difetto (*fault*): avviene durante l'esecuzione del programma e si manifesta in uno stato non corretto, il quale *può* portare, nel corso dell'esecuzione, a un fallimento;
- errore (*error*): indotto da uno sbaglio del programmatore che porta a un difetto il quale a sua volta può indurre un fallimento.

Esistono monitor che mettono in risalto la presenza di difetti, altri che ne effettuano la diagnosi (cioè cercano di scoprirne la causa), altri ancora che non solo forniscono informazioni sulla causa ma aiutano anche il sistema a risolvere la situazione di errore o riportandolo allo stato precedente o portandolo in uno stato corretto.

La descrizione del comportamento di un software avviene durante la stesura dei *requisiti software,* termine con il quale si intende la descrizione dei comportamenti esterni accettabili di un sistema, senza specificare il comportamento interno, cioè senza scendere in dettagli implementativi.

Le *proprietà software*, invece, sono relazioni tra gli stati di un programma e possono altresì essere viste come insieme di sequenze di stati, definendo principalmente quelle relazioni che portano a comportamenti esterni accettabili. Le proprietà associate al comportamento del software e i requisiti del problema sono descritti mediante un opportuno linguaggio di specifica.

Il *linguaggio di specifica* è utilizzato per la definizione delle proprietà da monitorare. Ne esistono vari tipi a seconda del livello di astrazione della

specifica stessa (se cioè essa è a livello di dominio del problema, della progettazione o dell'implementazione) e dell'espressività del linguaggio per quanto riguarda la capacità di definire il tipo delle proprietà e il livello di monitoring che si vuole adottare. Infatti, è possibile specificare se le proprietà debbano venir valutate a livello di programma, di moduli, di azione o di evento. Il linguaggio utilizzato per la specifica delle proprietà può basarsi sull'algebra, sugli automi a stati finiti o sulla logica o può essere un linguaggio funzionale, un linguaggio imperativo oppure orientato agli oggetti.

Lo scopo di un monitor è dunque quello di utilizzare le proprietà per scoprire i difetti prima che questi divengano fallimenti. Sempre secondo [20]

un monitor è un sistema che osserva un altro sistema e verifica se esso sia consistente con una data specifica.

Date una traccia di esecuzione di un programma e la specifica delle sue proprietà, il monitor controlla se tale traccia conservi le proprietà, studiando in questo modo solo la transazione *corrente*, piuttosto che tutte le possibili transazioni.

Definito $\mathcal{P}$ un programma in esecuzione costituito da un insieme di thread che eseguono in parallelo, in modo sequenziale o a intervalli, e $\sum_{\mathcal{P}}$ lo stato di un programma $\mathcal{P}$ in esecuzione, corrispondente allo stato dell'insieme di tutti i thread in esecuzione, la *traccia di esecuzione di un programma* $\mathcal{P}$ è la sequenza, eventualmente infinita, di stati del programma.

Una proprietà software ha la forma:

$$\mu \rightarrow \alpha$$

dove μ esprime una condizione in $\sum_{\mathcal{P}}^{i}$ che identifica lo stato in cui α deve continuare a valere. In altre parole, in ogni stato in cui μ vale *true* in $\sum_{\mathcal{P}}^{i}$, anche α deve essere *true* in $\sum_{\mathcal{P}}^{i}$; se α viene altrimenti valutato *false*, l'esecuzione corrente viene a trovarsi in uno stato non valido.

3.4 I componenti di un tool di monitoring

Secondo quanto detto sopra, un monitor è composto da due parti:

- un *osservatore* (observer) che valuta μ;
- un *analizzatore* (analyzer) che valuta α.

Quando l'analyzer scopre la violazione di una proprietà, cioè verifica ad esempio che α ha assunto valore *false* quando invece dovrebbe essere *true,* il monitor risponde in qualche modo: può iniziare una certa azione, come interrompere il programma, può avviare una routine di recovery o spedire i dati raccolti a un log.

Esiste poi una terza parte di solito esterna al tool di monitoring, l'*event-handler* che gestisce i risultati prodotti dal monitor, li comunica all'utente o a un altro sistema e di solito risponde in qualche modo alla violazione registrata.

L'event-handler si occupa della reazione che il monitor intraprende quando viene avvertita una violazione. Alcuni tipi di event-handler danno un unico tipo di reazione, indipendentemente dalla violazione riscontrata, altri invece lasciano che sia l'utente a scegliere l'azione da intraprendere. Un' ulteriore classificazione tra gli event-handler si basa sull'effetto che una certa azione da parte del monitor può avere sul programma in esecuzione. I monitor che si limitano a riportare le tracce di esecuzione o un log della violazione non hanno alcun effetto sull'esecuzione dell'applicazione target, a parte eventuali rallentamenti sulla velocità di esecuzione. Altri event-handler non proseguono nell'esecuzione finchè non avviene un'azione da parte dell'utente: appartengono a questa categoria i tool che utilizzano i breakpoint. Infine, gli event-handler più evoluti sono quelli che utilizzano la terminazione del programma o azioni di recupero dell'esecuzione, come il meccanismo delle eccezioni o i rollback, per rispondere a eventuali violazioni.

In Figura 3.3 è raffigurato un sistema composto dai tre elementi sopra descritti.

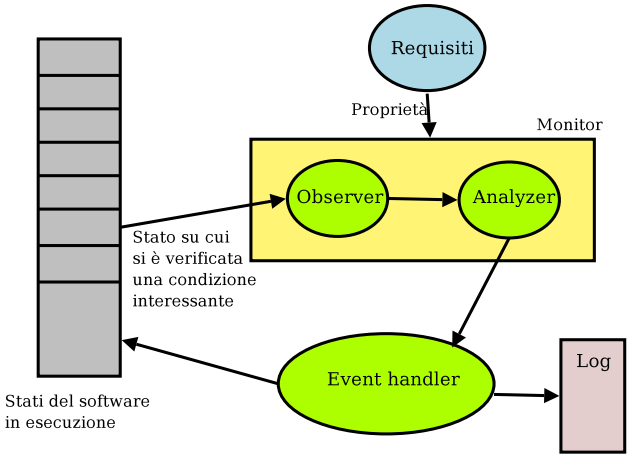


Figura 3.3: Una schematizzazione del monitoring

Come semplice esempio del funzionamento di un sistema di monitoring, supponiamo che un programma non permetta a due thread di accedere contemporaneamente alla risorsa condivisa A. La proprietà relativa viene espressa nel modo seguente: "Quando un thread accede ad A, il numero di thread al momento su A deve essere esattamente uno". Se la condizione μ è data dall'accesso ad A e α è dato dal numero di thread che deve essere esattamente uno, all'observer spetta il compito di rilevare che un thread ha appena fatto accesso alla risorsa condivisa A, cioè che si è verificata la condizioneμ e all'analyzer spetta verificare che α sia rispettata. Se il numero di thread su A è due, l'analyzer notifica questo evento che verrà successivamente gestito dall'event-handler.

Capitolo 4

Tecniche di monitoring

4.1 Classificazioni di monitoring

Oltre alla classificazione funzionale introdotta nel capitolo precedente, è possibile suddividere gli strumenti di monitoring in varie classi dipendenti da altrettanti fattori: le classi di proprietà software che possono essere specificate e applicate, l'infrastruttura richiesta, il supporto fornito dall'utente e il tipo di applicazione a cui il monitoring si applica. Le categorizzazioni più interessanti, però, avvengono in base al linguaggio di specifica, al tipo di monitoring e al tipo di event-handler come descritto in [20].

1. Il linguaggio di specifica delle proprietà si caratterizza in base al livello di astrazione e all'espressività, dal momento che è possibile definire proprietà design-based o implementation-based, ossia che specifichino proprietà a livello di progettazione e indipendenti da come avvenga l'implementazione oppure legate direttamente ai costrutti di programmazione. Inoltre, una proprietà può essere valutata a diversi livelli all'interno di un software: a livello di programma, di modulo, di espressione o di evento.

2. I vari tipi di monitoring possono essere classificati secondo la modalità di individuazione dei *monitoring point*, ossia dei punti di codice da cui

iniziare la procedura di monitoring. Il posizionamento dei monitoring point può avvenire in modo manuale o automatico con l'utilizzo di componenti specifici. Il monitoring può avvenire inline o offline, ossia nello stesso processo del programma monitorato o in un processo separato, di solito anche su una macchina diversa.

3. Gli event-handler si suddividono in base al livello di controllo, per cui esistono event-handler che reagiscono a tutte le violazioni riscontrate e altri che permettono di selezionare gli eventi in conseguenza ai quali reagire, e in base alla risposta data: alcuni lasciano all'utente il compito di rispondere in modo appropriato alla violazione, altri rispondono in maniera automatica (ad esempio con la terminazione del programma o con la gestione delle eccezioni), altri ancora noneffetuano alcuna azione e si limitano a registrare la violazione in un file di log.

In realtà, spesso la scelta di adottare uno strumento di monitoring piuttosto che un altro dipende da fattori contingenti, per cui si predilige uno strumento perché effettua il monitoring di applicazioni specifiche o perché è l'unico che opera su un certo sistema o con certe ristrettezze da un punto di vista dell'hardware. Ad esempio, esistono tool di monitoring che si rivolgono esplicitamente alle applicazioni distribuite o ai sistemi runtime.

4.2 Alcuni software fault monitoring

Se si pensa al comportamento dinamico di un programma secondo il concetto dello stato del computer (*computer state*), cioè secondo i contenuti della memoria a un certo momento dell'esecuzione, si può vedere il monitoring come il controllo della validità di ciascuno stato [19]. Ciascun passo in avanti nell'esecuzione comporta il passaggio da uno stato all'altro attraverso successive modifiche del contenuto di una o più locazioni di memoria e l'esecuzione del programma viene definita come sequenza di questi passaggi detti transizioni

di stato (*state transition*), a cui sono associati le *asserzioni*[1] e i vincoli di integrità. I primi approcci al software fault monitoring (Anna, Annotation PreProcessor, Eiffel) richiedevano che lo sviluppatore inserisse le asserzioni nei punti strategici del codice. Di seguito viene proposta una breve rassegna, tratta da [20, 19], senza pretese di esaustività, dei primi strumenti di monitoring e di altri più recenti progettati per le applicazioni Java.

4.2.1 Anna, Concurrent Anna e Anna Consistency Checking System

Anna [19] utilizza un'estensione dei costrutti del linguaggio Ada per specificare le asserzioni. Spetta allo sviluppatore specificare, sotto forma di commenti formali (anche detti *annotazioni*) che non sono altro che espressioni booleane e parole chiave, le asserzioni da applicare a oggetti, tipi, espressioni e moduli. E' possibile anche effettuare brevi computazioni esternamente al programma per controllare lo stato delle asserzioni. Ciascun commento formale viene poi sostituito con chiamate a opportuni moduli di controllo.

Concurrent Anna [19] si basa sullo stesso approccio al monitoring basato sulle asserzioni di Anna, riuscendo inoltre a eseguire i moduli associati ai commenti formali in modo concorrente.

Anna Consistency Checking System [20], anch'esso derivato da Anna, genera funzioni di controllo in corrispondenza di ciascuna annotazione. Le funzioni di controllo vengono inserite laddove possano verificarsi delle inconsistenze con i commenti formali, come chiamate di procedure, conversioni di tipo o assegnamenti, per effettuare verifiche di consistenza in modo concorrente rispetto all'esecuzione del programma sottostante. Qualora si verifichino inconsistenze, spetta all'event-handler decidere se ignorarle, annotarle nel log o terminare l'applicazione monitorata a seconda della gravità di queste.

[1] Le asserzioni, una sottoclasse dei vincoli d'integrità, sono costrutti che controllano l'esecuzione dinamica dei programmi verificando l'integrità delle assunzioni su cui si basa il programma a ogni esecuzione.

4.2.2 Annotation PreProcessor (APP)

Come in Anna, anche in Annotation PreProcessor [20] lo sviluppatore che lavora sul linguaggio C aggiunge le asserzioni nei punti in cui è necessario effettuare i controlli e a livello del modulo, per mezzo delle pre-condizioni e delle post-condizioni. In questo modo APP traduce un programma C con uno equivalente che presenta le asserzioni inserite rispetto all'originale e, per mezzo del pre-processore, riesce a instrumentare il codice con espressioni di controllo. Lo sviluppatore può anche inserire un livello di severità, che indichi l'importanza di un'asserzione e determinare se questa verrà poi verificata a runtime, e decidere quale sia la risposta all'eventuale violazione dei vincoli..

4.2.3 Eiffel

Eiffel [5] fornisce una metodologia di supporto all'analisi, alla progettazione e alla fase di sviluppo del software orientato agli oggetti con la definizione di pre e post condizioni, di asserzioni, di invarianti sui costrutti `while`, di classi e di routine. Le asserzioni che specificano le invarianti di una classe sono integrate nel linguaggio stesso ed estratte automaticamente. In questo modo, con l'introduzione del *Design by Contract*, Eiffel facilita la creazione del software secondo un approccio sistematico con procedure per il runtime software fault monitoring. Design by Contract è la metodologia secondo cui i progettisti software devono definire delle specifiche precise ed esaustive per le interfacce dei componenti basandosi sulla teoria dei tipi di dato astratti e sulla metafora concettuale del contratto d'affari, secondo cui ciascun componente deve garantire determinate funzionalità e aspettarsene altre in ritorno ottenendo un beneficio da entrambe le parti coinvolte. I concetti a cui si attinge fortemente sono la comprensione e il corretto utilizzo di meccanismi potenti quali l'ereditarietà tra oggetti, il polimorfismo, la gestione delle eccezioni e la documentazione automatica del software, partendo da basi quali la verifica e la specifica formale e la logica di Hoare. La ricerca di Design by Contract è tuttora attiva e rivolta soprattutto alle problematiche introdotte dalla concorrenza e dai sistemi distribuiti.

4.2.4 ALAMO

ALAMO (A Lightweight Architecture for MOnitoring) [15, 24, 20], progettato per applicazioni C e Icon, riduce fortemente le difficoltà presenti nello sviluppo di tool di analisi dinamica, come i profiler special purpose o i bug-detector, instrumentando il codice C del programma da monitorare in modo automatico, secondo le direttive dell'utente specificate nella configurazione del tool. I tool di analisi dinamica sono utilizzati in molte fasi dello sviluppo del software, compresi la scrittura del codice, il testing e la manutenzione e il costo dello scrivere tool simili può essere piuttosto alto. Sebbene non sia presente un linguaggio di specifica formale ad alto livello, il linguaggio che ALAMO utilizza per definire la configurazione permette di dichiarare quali siano le attività che devono essere percepite come eventi da analizzare. ALAMO infatti si serve di CCI (Configurable C Instrumentation) come pre-processore che identifica i punti su cui effettuare il monitoring e inserisce gli eventi nel codice sorgente del programma *target*, eseguito poi dal componente di ALAMO detto Execution Monitor (EM). Al termine dell'esecuzione, le informazioni sono disponibili sotto forma di report degli eventi, che l'utente può rendere più specifico con l'applicazione di predicati.

4.2.5 Java PathExplorer

Java Path Explorer (JPaX) [20] è uno strumento general-purpose per il monitoring di programmi Java dall'esecuzione sequenziale e concorrente, utilizzando specifiche dei requisiti formali scritte in logica temporale lineare o nel linguaggio di specifica algebrico Maude. JPaX instrumenta il byte code di Java per trasmettere un flusso di eventi rilevanti al modulo di osservazione il quale effettua due tipi di analisi: un monitoring logic-based (che consiste nel verificare gli eventi a fronte delle specifiche dei requisiti ad alto livello) e un'analisi error pattern (ossia che effettua la ricerca degli errori di programmazione a basso livello). Mentre Maude è utilizzato per confrontare le tracce di esecuzione con le specifiche, l'analisi a runtime degli error pattern, invece, ricerca eventuali errori, come la presenza di stalli, nelle tracce di esecuzione, anche qualora questi errori non compaiano esplicitamente nelle tracce.

4.2.6 Monitoring and Checking

Monitoring and Checking (MaC) [20] fornisce un'architettura per il monitoring di sistemi real-time sviluppati in Java. Invece di utilizzare le asserzioni inserite nel linguaggio, l'utente, in fase di progettazione, crea uno script di monitoring che definisce gli eventi, produce un riconoscitore che associa le informazioni sugli stati agli eventi e instrumenta infine il codice al fine di recuperare le informazioni rilevanti sugli stati del programma. MaC si serve di un componente che agisce da filtro e che è responsabile dell'estrazione del valore delle variabili del programma e del loro invio al riconoscitore di eventi, il quale comunica le informazioni da controllare a un run-time checker.

Sebbene il monitor e il checker di MaC possano lavorare in modo concorrente, essi non hanno bisogno di agire insieme né di coordinarsi. Il monitor può inoltrare le informazioni al checker in un file e questi può verificarle in un momento successivo. Il riconoscitore di eventi e il filtro sono anch'essi due processi separati, sebbene tutti i componenti di MaC debbano risiedere sulla stessa macchina. MaC è spesso utilizzato per scoprire eventuali difetti nell'esecuzione delle computazioni numeriche e nelle sequenze di eventi.

4.2.7 Monitoring-Oriented Programming

Monitoring-oriented programming (MoP) [20] permette di specificare le proprietà formali di un linguaggio di programmazione oggetto del monitoring (linguaggio di programmazione target) e, da queste, generare codice per il monitoring. Il codice generato deve contenere i seguenti componenti: dichiarazioni, inizializzazione, corpo del monitoring, condizione di successo e condizione di fallimento. L'utente annota poi i punti nel programma in cui dovrebbe essere posizionato il codice del monitoring, insieme alle informazioni sulla logica usata. Prima della compilazione, il generatore di codice prende le specifiche formali e crea il codice per il monitoring nello stesso linguaggio di quello target. MoP, di cui una versione Java-MoP è stata implementata come applicazione client-server, riesce a gestire le violazioni eseguendo opportuno codice di ricovery, visualizzando e inviando messaggi o lanciando eccezioni.

4.2.8 JaVis

Il tool JaVis [11] descrive le relazioni esistenti tra i componenti di un programma Java generando tracce descrittive sulle chiamate dei metodi al fine di individuare deadlocks. La tecnica utilizzata prevede l'utilizzo dell'interfaccia JDI di JPDA6.1, che permette di monitorare i thread con il meccanismo della sospensione. JaVis si propone di comprendere l'evoluzione della concorrenza tra thread in modo da facilitarne il debugging e individuare eventuali deadlocks, automaticamente rilevati e analizzati attraverso la costruzione di grafi che mettono in luce la presenza di cicli e di *lock* su risorse sincronizzate. I risultati sono poi visualizzati sotto forma di UML sequence e collaboration diagrams.

Parte II

Un tool per il monitoring architetturale

Capitolo 5

Un tool di monitoring - EXaM

5.1 Gli obiettivi di EXaM

Il risultato del monitoring su un sistema complesso può risultare in una quantità enorme di dati che ne descrivono il comportamento. Affinchè questi dati siano valutati e sottoposti a un *event-handler* che li analizzi e li gestisca, è necessario avvalersi di uno strumento che permetta di selezionare automaticamente i più rilevanti ai fini del monitoring tra tutti quelli recuperati. Per sopperire a questa esigenza, il tool qua presentato, avvalendosi dell'interfaccia JDI dell'architettura JPDA di Sun, introduce una tecnica che permette di generare solo le tracce ritenute interessanti, senza dover ricercare *a posteriori,* tra tutte le informazioni ricavate, i risultati che ci permettano di valutare il comportamento del sistema.

EXaM si rivolge ai sistemi a componenti distribuiti sviluppati in Java con tecnologia RMI, definiti da un'architettura software che ne rappresenti la configurazione. EXaM effettua il monitoring del sistema generando solo le tracce di esecuzione ritenute rilevanti a livello architetturale e derivando dati che descrivono le proprietà dei connettori, ossia le relazioni tra gli elementi strutturali che compongono il sistema. In particolare, la definizione del comportamento che il sistema deve assumere in fase di esecuzione e la configurazione degli elementi che lo compongono sono rappresentati da un *file di configurazione* in cui vengono descritti gli elementi strutturali del mo-

nitoring, ossia gli host sui quali verrà eseguita l'applicazione target. Questo file di configurazione viene dunque a rappresentare il punto d'incontro tra l'astrazione del sistema rappresentata dalla definizione dell'architettura software e la sua implementazione da parte dei componenti distribuiti sui vari host.

Le tracce generate vengono ottenute monitorando le interazioni all'interno di ogni componente e quelle tra i componenti distribuiti (vedi in Figura 5.1), vale a dire le interazioni locali e quelle remote. Anche le interazioni interne a un componente possono essere viste come interazioni tra componenti, se per componente si intende una classe, un package o un'interfaccia.

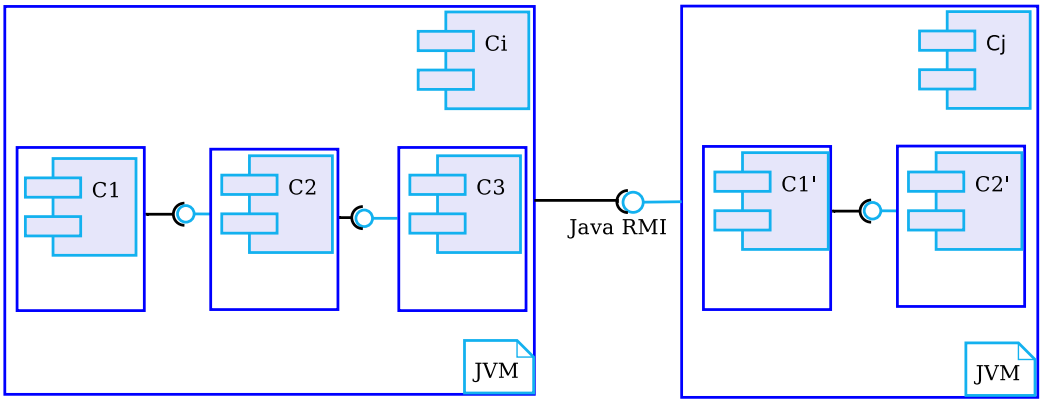


Figura 5.1: La comunicazione inter-componente

Ogni componente, quindi, produce i propri risultati riguardo il monitoring dei moduli interni e le chiamate remote agli altri componenti. Le tracce delle chiamate remote di ciascun componente vengono poi spedite al master che si occupa di combinarle insieme e tracciare quindi uno schema delle interazioni a livello di sistema. I risultati hanno sempre la forma di tracce di esecuzione in formato XML. In queso modo, EXaM effettua anche un duplice controllo a livello di unità e di integrazione: il primo in base alle tracce prodotte dai componenti locali, il secondo sulle tracce prodotte dalle interazioni remote. E' possibile anche non richiedere alcuna traccia, ottenendo in questo modo solo l'esecuzione dell'applicazione target su un componente.

Le invocazioni di cui si ottengono le tracce sono quelle che avvengono tra componenti diversi, in modo trasparente rispetto al fatto che essi siano

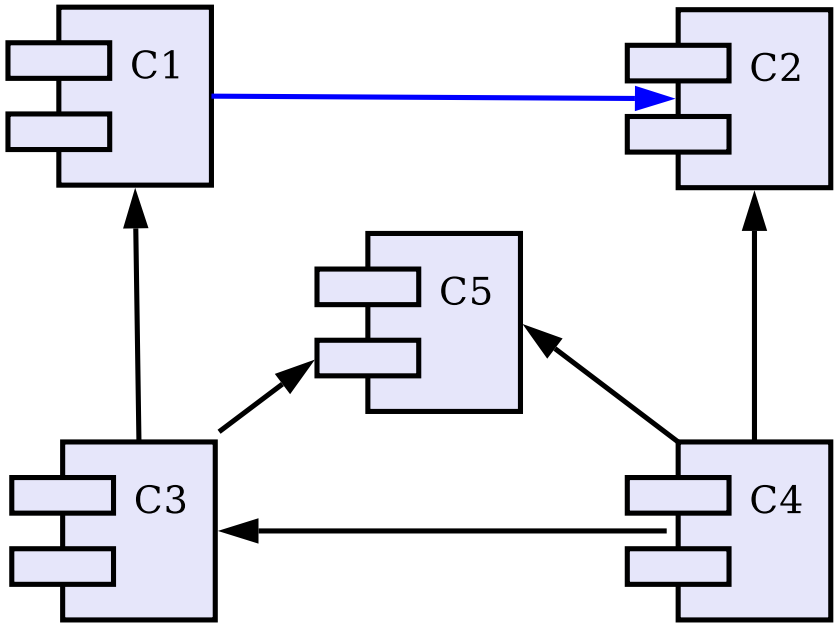


Figura 5.2: Le invocazioni tra componenti

locali o remoti. Di tutte le possibili invocazioni inter-componente, l'utente di EXaM, attraverso un'opportuna compilazione del file di configurazione (di cui si dà una descrizione dettagliata nel paragrafo 7.1.1), seleziona quelle relative ai componenti che vuole monitorare. Per cui, dato un sistema schematizzato come in Figura 5.2, se nel file di configurazione si specificano i componenti *C1* e *C2* l'unica invocazione tracciata sarà quella disegnata in blu.

Le tracce vengono prodotte secondo una tecnica *debugger-like* e con l'utilizzo dell'interfaccia JDI di JPDA (descritta nel paragrafo 6.1). Tale tecnica risulta essere meno invasiva rispetto ad altre quali l'instrumentazione del codice o del byte-code e permette di scendere a vari livelli di dettaglio nella generazione delle tracce. Sebbene si sia scelto di non effettuare analisi delle performance né di studiare l'occupazione della memoria o altri parametri di profiling, tuttavia il tool si presta facilmente a integrazioni o modifiche per ampliare le funzionalità offerte. Esso è infatti sviluppato in modo totalmente modulare, così da potersi prestare a eventuali future modifiche sul metodo di monitoring e su quello di visualizzazione e analisi dei risultati (vedi 5.6).

EXaM è inoltre in grado di limitarsi alla semplice esecuzione di appli-

cazioni, favorendo in questo modo l'automatizzazione di casi di test in ambiti specifici e la produzione di dati da sottoporre a un tool di validazione. Da questa prospettiva, dunque, EXaM ottiene la descrizione del sistema da monitorare, più o meno completa a seconda di quanto l'utente richieda in fase di configurazione, insieme alla sua istanza, ossia insieme a un'esecuzione completa e realistica su un sistema distribuito. I rallentamenti prodotti a seguito delle sospensioni operate sui thread non compromettono comunque il funzionamento globale dell'applicazione da monitorare e, dal punto di vista di un eventuale sistema di testing che adoperi i risultati prodotti da EXaM, possono essere assimilabili a uno dei tanti fattori presenti in un sistema reale.

5.2 Le origini e il contesto

EXaM nasce come continuazione di Teseo, un precedente progetto di analisi dinamica di applicazioni Java, avviato nel 2005 dall'istituto ISTI del CNR e sviluppato da Alessia Bardi come tesi di tirocinio. Teseo monitorava un'applicazione Java verificando le interazioni a runtime tra le componenti che la costituiscono e generando tracce di esecuzione descrittive del comportamento del sistema. Teseo limitava però il suo campo di applicabilità alle Java Virtual Machine locali, dal momento che il tool di monitoring e l'applicazione target dovevano necessariamente risiedere sullo stesso host, e non avviava l'esecuzione senza prescindere dal monitoring stesso. Si è dunque scelto, in primo luogo, di espandere Teseo per poter analizzare sistemi distribuiti e poter eseguire l'applicazione target su un altro processore o su un numero indefinito di processori; inoltre si è deciso di separare concettualmente il monitoring dall'esecuzione dell'applicazione, al fine di renderli indipendenti l'uno dall'altra. Ciò che EXaM ha mantenuto di Teseo è il concetto di componente come oggetto del monitoring: se un componente è un'entità *black-box*, con il quale è possibile interagire per mezzo di interfacce pubbliche, un'istanza di un componente è un artefatto riusabile e sostituibile da un altro con le stesse funzionalità. L'interesse verso l'istanza di un componente, e quindi l'oggetto del monitoring, è dunque dovuto alle sue funzionalità e non alla sua implementazione o al contesto in cui esso sia inserito. Nell'ottica di un'appli-

cazione distribuita, l'oggetto del monitoring non si limita solo ai componenti costituenti un'applicazione, ma riguarda anche i componenti costituenti un sistema distribuito.

Il nuovo strumento si colloca all'interno di PLASTIC [13], un progetto più ampio il cui obiettivo è lo sviluppo di una piattaforma per la distribuzione di servizi wireless in un ambiente dinamico ed eterogeneo che supporti lo sviluppo di applicazioni innovative dal punto di vista della qualità offerta. La problematica della validazione di sistemi di questo tipo è dunque centrale nella visione di PLASTIC e richiede particolare attenzione soprattutto a fronte della necessità di fornire nuovi strumenti e nuove metodologie che permettano a strategie diverse di testing e di monitoraggio di interagire e che conciliino gli aspetti di interoperabilità tra i vari livelli applicativi.

5.3 Le caratteristiche

La struttura di EXaM è assimilabile a quella delle architetture eterogenee che prevedono più di uno stile architetturale, come descritto nel paragrafo 2.1.7. EXaM infatti rientra nelle categorie dei sistemi distribuiti, dal momento che è strutturato in componenti che interagiscono tra loro su una rete. In particolare, il modello di comunicazione seguito è il *master/slave* (anche detto *primary/secondary*), in cui un processo controlla gli altri attraverso una serie di direttive volte a guidare il flusso e l'elaborazione dei dati (maggiori dettagi nel paragrafo 6.4.2). La comunicazione, seppure essa avvenga anche dallo slave al master o tra i vari slave, è sempre indotta dal processo principale, ossia il master. Data però la natura della comunicazione all'interno di EXaM, volta a eseguire l'esecuzione e il monitoring delle applicazioni distribuite, il processo master detiene una serie di dati e di informazioni di cui alcune precluse ai vari slave. Ad esempio, mentre il componente A non conosce ciò che viene monitorato all'interno del componente B e viceversa, il master è a conoscenza di ciò che avviene in entrambi i componenti : questi ultimi devono solo coordinarsi con il processo principale attraverso chiamate remote e, se l'applicazione oggetto del monitoring lo richiede, comunicare tra di loro per mezzo di altrettante chiamate RMI. Da questa prospettiva è

possibile riconoscere le principali caratteristiche dei sistemi data-centered e riconoscere nel componente master , seppur con le dovute semplificazioni, ul repository con le informazioni principali analizzate ed elaborate dagli altri.

In rispetto ai principi della Software Architecture, EXaM presenta le seguenti caratteristiche:

- *estendibilità*: per la sua natura inerentemente modulare, EXaM si presta a un costante aggiornamento della componente di monitoring. E' possibile infatti sostituire il componente che effettua il tracing delle relazioni senza modificare il resto dell'applicazione, con la possibilità di sostituirlo con un altro dalle diverse funzionalità;

- *unione in un unico tool di funzionalità di solito svolte in più di un ambiente*: EXaM fornisce un unico ambiente in cui è possibile muoversi tra requisiti progettuali, monitoring e visualizzazione delle tracce prodotte. Data la possibilità di utilizzare EXaM per la sola esecuzione delle applicazioni, inoltre, esso si concilia in modo natutrale con uno strumento di testing dei risultati prodotti.

- *linguaggio di specifica*: EXaM non introduce né adotta alcun linguaggio di specifica. I componenti da monitorare e la configurazione dell'architettura distribuita sono infatti incorporate nel tool stesso, più precisamente nel file di configurazione con cui EXaM viene avviato. Spetta all'utente che lo redige specificare i dettagli del monitoring, ossia se EXaM debba limitarsi a tracciare le interazioni tra i componenti interni o quelle tra i componenti esterni, vale a dire le interazioni tra i vari host distribuiti in rete. Anche l'architettura distribuita viene infatti articolata in base al file di configurazione, attraverso la specifica delle macchine su cui verranno attivati gli opportuni strumenti per il monitoring e dei componenti da eseguire su ciascuna macchina.

- *raccolta delle informazioni*: spetta all'utente stesso scegliere quali componenti monitorare a livello di metodi e di packages. Ciò personalizza in modo molto potente le funzionalità del tool e permette di porre un

filtro sui risultati ottenuti, facilitando lo studio e l'analisi di tracce già selezionate secondo la configurazione specificata.

5.4 Il funzionamento

Di seguito si descrive brevemente il funzionamento di EXaM. I dettagli implementativi verranno trattati ampiamente nei capitoli 8 e 9.

5.4.1 Il file di configurazione

Come già descritto sopra, il file di configurazione di EXaM, in formato XML, descrive concretamente il sistema, ossia ciò che l'architettura software delinea in modo astratto. Con questo file si determina una corrispondenza univoca tra l'architettura software e il sistema reale di componenti realizzato su una rete con gli host coinvolti nel monitoring.

L'utente che compila il file di configurazione deve avere una conoscenza del sistema e di cosa si vuole monitorare. Al momento della compilazione, inoltre, è necessario conoscere l'esatto mapping tra i componenti descritti nell'architettura software e gli host sulla rete, in modo da eseguire e monitorare ogni applicazione target sul componente corretto.

Ciò che si richiede è l'individuazione della struttura e delle decisioni chiave che riguardano il modo in cui il software deve funzionare in termini di comportamenti attesi e di tracce prodotte. In particolare è richiesto di:

- *individuare la struttura del sistema in termini di componenti*: significa determinare gli host coinvolti nel monitoring e le applicazioni target da eseguire su ciascuno di essi;
- *specificare le connessioni tra i componenti di cui si vuole ottenere il monitoring*: consiste nell'individuare i componenti (packages e interfacce) e le invocazioni di si vuole ottenere le tracce di esecuzione;
- *supportare in modo appropriato il processo di monitoring*: consiste nell'occuparsi di vari dettagli tecnici, come individuare le porte attive, le

directory in cui produrre i risultati e i parametri della Virtual Machine necessari per eseguire e avviare il monitoring delle applicazioni target.

EXaM fornisce anche un supporto grafico per la compilazione del file di configurazione, qualora essa risultasse ostica o troppo a basso livello per utenti inesperti.

5.4.2 Lo schema master - slave

EXaM (vedi Figura 5.3) esegue ed effettua il monitoring di un sistema distribuito su una rete di processori. La configurazione del master e degli slave risulta totalmente centralizzata, poiché, come già spiegato, è il master a predisporre, sulla base delle direttive dell'utente, quali componenti gireranno sui vari host della rete. Una tale struttura facilita il recupero e la gestione delle informazioni riguardanti le relazioni inter-componente e comporta solo l'installazione dei programmi che implementano lo slave su ciascun host, al momento di configurare la rete ospite. Ciascun monitoring, quindi, richiede solo l'avvio di uno o più slave, a seconda della complessità dell'applicazione oggetto dell'analisi, lasciando in attesa di una richiesta di esecuzione gli altri slave non coinvolti.

Con il parsing del file XML di configurazione, il master recupera tutte le informazioni necessarie ad eseguire l'applicazione target e ad effettuarne il monitoring: il numero e l'indirizzo dei nodi coinvolti e, per ogni nodo, il componente da eseguire e le relazioni da tracciare. Il master trasmette poi a ogni nodo, attraverso la metodologia RMI, le informazioni specifiche, avvia i rispettivi processi su ciascun slave e aspetta i risultati dell'elaborazione.

Alla chiamata del master, ciascun slave risponde leggendo le informazioni che lo riguardano, ossia tutte le informazioni relative al componente da eseguire, e provvedendo a scaricare il codice eseguibile della componente target, utilizzando un listener messo a disposizione dal master. Con questi dati, lo slave può eseguire il monitoring del componente, accedendo a livello della Virtual Machine in esecuzione. Per le interfacce e i packages di cui si è fatta richiesta, vengono derivate le tracce, sotto forma di file XML e compresse in

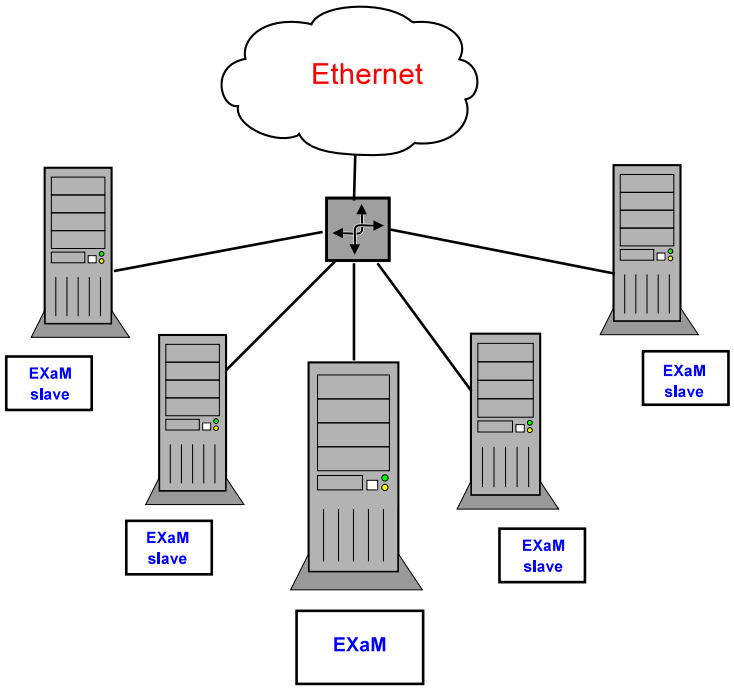


Figura 5.3: La struttura di EXaM

un unico file .zip, infine spedite al master perché le elabori e le convogli verso un'entità esterna predisposta ad analizzarle.

Il funzionamento di EXaM, suddiviso secondo il master e lo slave, è dettagliatamente descritto da un punto di vista tecnico in 7.

EXaM, eseguendo l'applicazione oggetto del monitoring, verifica che essa sia stata concepita e sviluppata correttamente non solo producendo tracce che verranno analizzate da un tool specifico, ma intervenendo in modo assolutamente non invasivo nell'avvio di ciascun componente, favorendo dunque l'emergere di eventuali difetti nel codice. Poniamo, per esempio, che l'applicazione da monitorare sia una semplice applicazione RMI costituita da un server, app1, che implementa il metodo `Hello()` e un client, app2, che richiama tale metodo in remoto. EXaM esegue il codice di app1, fornito come file `app1.jar` (per i dettagli sulla compressione di un'applicazione in un unico file jar, vedi Appendice A), sull'host A ed esegue poi `app2.jar` sull'host B senza un particolare ordine e soprattutto senza attendere che app1.jar sia correttamente in esecuzione e in attesa di chiamate remote. Tale compor-

tamento mira a far emergere eventuali e frequenti errori nello sviluppo di un'applicazione RMI:

1. il client non gestisce il caso in cui il server remoto non sia attivo perché momentaneamente fuori uso o non ancora avviato;
2. il client e il server utilizzano stub validi solo nel classpath locale, ossia definiti con percorsi del tipo `file://` e senza un apposito codebase che si registri correttamente sull'rmiregistry;
3. il client contiene riferimenti all'interfaccia remota, senza un'appropriato meccanismo che riconduca all'implementazione della classe.

Analogamente, EXaM mira a evidenziare eventuali malfunzionamenti mettendo a disposizione le tracce delle chiamate remote e le tracce delle chiamate interne tra componenti diversi, sebbene la compilazione del file di configurazione possa risultare onerosa dal momento che richiede una conoscenza dettagliata dell'applicazione da analizzare.

5.5 Risultati

I risultati prodotti da EXaM consistono in primo luogo in una serie di file XML prodotti da ciascun slave secondo quanto specificato nel file di configurazione. Essi contengono informazioni sulle chiamate interne a ciascun componente e sulle chiamate remote verso gli altri componenti. Infine viene prodotto anche un altro file XML contenente tutte le interazioni, a livello di sistema software, tra gli oggetti distribuiti, che hanno coinvolto componenti di cui si era richiesto di tracciare il monitoring. Infatti, tutti i risultati sono vincolati al file di configurazione, dal momento che si è scelto di monitorare solo le chiamate tra i componenti (interfacce, packages) indicati nel documento XML compilato dall'utente, comprendendo tra queste anche le interazioni che avvengono tra gli host coinvolti nel monitoring. Come già accennato sopra, EXaM considera ciascun package all'interno di un'applicazione come un componente vero e proprio, per cui la generazione delle tracce

'interne' a un componente o della tracce tra componenti costituenti un'applicazione distribuita sono in realtà assimilabili allo stesso concetto di relazione inter-componente.

5.5.1 Tracce delle chiamate interne a ciascun componente

EXaM traccia le interazioni che, all'interno di un componente in esecuzione su un host, avvengono tra packages diversi, ossia traccia le chiamate a metodi in cui l'oggetto chiamante appartiene a un package diverso da quello dell'oggetto invocato. EXaM, infatti, sfrutta fortemente l'organizzazione in packages caratteristica dei linguaggi orientati agli oggetti. La motivazione che induce a generare le tracce che avvengono tra componenti diversi deriva dalla considerazione di un package alla stregua di un componente, per cui tracciare le interazioni tra componenti diversi si risolve nel tracciare le interazioni tra interfacce appartenenti a packages diversi.

La generazione delle tracce avviene risalendo al thread che ha invocato i metodi, distinguendo così tra caller e callee, ossia tra il componente che ha avviato la chiamata e il componente a cui appartiene il metodo chiamato. Si individua anche la classe di appartenenza del caller e del callee e si tiene traccia del tipo e del valore dei parametri dei metodi invocati.

Per ogni thread coinvolto nelle interazioni tra componenti viene generato un file XML con le informazioni sulle chiamate effettuate ai vari metodi, distinguendo se stiamo entrando in un metodo o se ne stiamo uscendo. Viene poi creato un altro file XML con il valore delle variabili coinvolte all'ingresso di ogni metodo chiamato dal thread monitorato. Inoltre, è possibile anche specificare un *breakpoint* (attraverso la specifica della classe e della linea di codice) nel file di configurazione di cui si desideri conoscere il valore delle variabili coinvolte, proprio come avviene nei classici strumenti di debugging.

5.5.2 Tracce delle interazioni tra gli oggetti distribuiti

Alla fine di ogni esecuzione, EXaM genera un file XML con le tracce delle interazioni tra i componenti distribuiti su ogni host. Questo documento, oltre a comparire a fianco delle altre tracce, viene visualizzato nell'ultimo pannello, chiamato `Traces,` dell'interfaccia di EXaM (per la quale vedi il paragrafo 7.3). Per la compilazione di questo file, EXaM si avvale dei file generati da ogni slave che riportano le tracce di tutte le chiamate remote effettuate da ciascun host. In questo modo, avvenuta la terminazione del monitoring e rientrati tutti i thread avviati sugli slave, è possibile ordinare le tracce remote e avere una visuale a livello di sistema dell'andamento delle chiamate tra componenti diversi. Per i dettagli sulla generazione di questi file rimandiamo al capitolo 9 e al semplice caso di studio presentato.

5.6 Future works

Essendo EXaM estremamente modulare e operante in un'area in cui si richiedono sempre modifiche e campi di applicabilità diversi, si possono pensare a molte modifiche che apporterebbe cambiamenti e miglioramenti sostanziali all'intero sistema. Una probabaile evoluzione di EXaM potrebbe in ogni caso avvenire in risposta a nuovi requisiti. Di seguito vengono proposti alcuni suggerimenti di ricerca verso cui è possibile orientarsi a partire dallo strumento presentato in questa sede.

1. **Linguaggio di specifica:** come accennato in 5.3, il linguaggio di specifica di EXaM è modellato all'interno del tool stesso. Tuttavia, si potrebbe pensare a un tipo di linguaggio diverso, separato dal file di configurazione, che permetta di definire ciò che si vuole monitorare in modo indipendente dalla configurazione dell'architettura distribuita.

2. **File di configurazione**: ad oggi il file di configurazione richiede un'approfondita conoscenza dell'architettura distribuita nonchè della struttura interna dei componenti che si intende monitorare. Sebbene EXaM fornisca già uno strumento grafico che assiste nella compilazione del

file, potrebbe risultare utile ripensare ai dati che occorre fornire per il setting del monitoring nell'ottica dell'utente e di ciò che gli si richiede.

3. **Visualizzazione dei risultati**: EXaM convoglia le tracce generate verso un'entità esterna con il compito di verificare i risultati. La presenza di un tale strumento all'interno di EXaM, insieme a una funzione di visualizzazione delle tracce non solo come file XML ma anche per mezzo di diagrammi UML, valorizzerebbe le potenzialità dello strumento di validazione, rafforzando ulteriormente le capacità di integrare in un unico strumento tutte le funzioni basilari per il monitoring (così come già accennato in 5.3).

4. **Tipi di applicazioni target**: EXaM traccia le interazioni che avvengono tra i componenti di un applicazione distribuita sviluppata secondo la tecnologia RMI. Tuttavia sarebbe utile estendere la categoria delle applicazioni target, comprendendo anche le applicazioni client-server, le connessioni via socket e più in generale tutte le comunicazioni di rete, agendo a livello di canale TCP/IP.

5. **Decentralizzazione**: in un'ottica di maggiore automatizzazione dei processi di monitoring e di validazione, il passo successivo rispetto a ciò che si è raggiunto con EXaM è indubbiamente la capacità di riuscire a configurare un'architettura distribuita che supporti la generazione delle tracce nel modo più indipendente e leggero possibile, senza l'eccessiva centralizzazione del controllo in un unico processo. Ciò comporta la capacità di eseguire anche gli slave da remoto così come si ottiene l'esecuzione dell'applicazione target con la sola indicazione dell'indirizzo in cui sia possibile recuperare il codice. In questo modo, l'unica configurazione che si ritiene necessaria è una rete senza firewall e con i permessi necessari affinchè il processo principale possa installare tutti i componenti coinvolti nel monitoring.

6. **Analisi dei thread**: sebbene ciascun thread venga sviscerataamente analizzato in ogni suo metodo e persino nelle variabili in ingresso,

sarebbe interessante tracciare anche la concorrenza e le sincronizzazioni tra thread, analizzando ciascuna chiamata di `wait`(), `notify()` e `notifyAll()`, combinando le nuove tracce con quelle derivate dalle interazioni tra componenti.

Capitolo 6

Background tecnico

6.1 Java Platform Debugger Architecture

Java Platform Debugger Architecture (JPDA) [8] fornisce un'architettura completa per la creazione di strumenti che effettuino debugging e profiling di applicazioni Java, anche distribuite, remote e cross-platform, senza doversi occupare degli aspetti relativi alla piattaforma, quali il sistema operativo, l'hardware o l'implementazione della Virtual Machine. Con questo approccio, è possibile analizzare i programmi durante la loro esecuzione e osservarli da una prospettiva che sarebbe altresì negata in un'analisi statica.

Sviluppata con l'uscita di Java 2 SDK, JPDA è composta dalle due interfacce JVMDI e JDI, dal protocollo JDWP e dai due componenti software, il front-end e il back-end.

Come illustrato in Figura 7.1, la struttura che la compone è costituita dai seguenti elementi:

- il **debuggee** - il processo su cui eseguire il debugger. Il debugge a sua volta è composto da

  - la **Virtual Machine** che esegue l'applicazione oggetto del debugger *(*detta *applicazione target*) e implementa la Java Virtual Machine Debugger Interface (JVMDI).

  - il **back-end** del debugger, responsabile delle richieste della componente front-end del debugger verso il debuggee e le risposte di quest'ultimo alle richieste del fornt-end, incluse le notifiche di eventi. Le comunicazioni tra back-end e front-end avvengono attraverso il communications channel e utilizzando il Java Debug Wire Protocol, mentre quelle tra back-end e VM sfruttano la Java Virtual Machine Debug Interface (JVMDI).

- un **canale di comunicazione** tra il debugger e il debuggee. JPDA non specifica alcun meccanismo di trasporto ed è dunque possibile utilizzare socket, memoria condivisa o canali seriali; solamente il formato e la semantica del flusso serializzato di bit sono dettato dal Java Debug Wire Protocol (JDWP).

- il **debugger**, composto da

  - il **front-end**, ossia l'implementazione dell'interfaccia ad alto livello Java Debug Interface (JDI) che utilizza le informazioni veicolate dal JDWP;
  - un'**interfaccia utente** (UI) del debugger a carica degli utenti di JPDA, poichè non implementata.

Una simile struttura 'multi-layer' permette approcci differenziati a seconda di ciò che si voglia sviluppare e della confidenza che si ha con gli aspetti più vicini all'implementazione delle singole Virtual Machine.

Di seguito sono illustrate le interfacce JVMDI e JDI e il protocollo JDWP. Poichè EXaM fa uso esclusivamente di JDI, di questa si tratterà diffusamente, mentre di JVMDI e JDWP si tracceranno solo le linee generali.

6.1.1 Il protocollo JDWP

Java Debug Wire Protocol è il protocollo utilizzato per le comunicazioni tra il debugger la target Virtual Machine. Sebbene il suo utilizzo sia del tutto opzionale e addirittura non implementato in alcune versione di Java SDK,

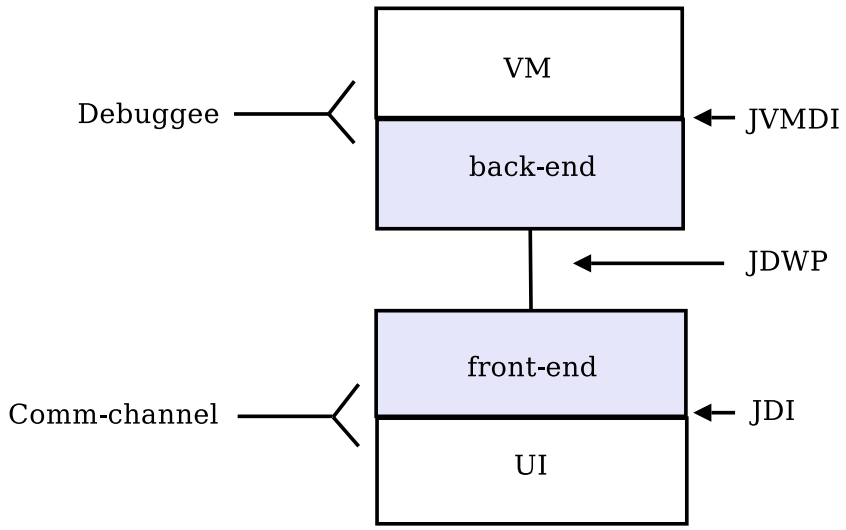


Figura 6.1: L'architettura di JPDA

esso permette di eseguire il debugging di un processo diverso da quello del debugger ma residente sullo stesso computer o addirittura di un processo su un computer remoto, specificando solo il formato e il layout dei dati e accettando vari meccanismi di trasporto.

JDWP stabilisce una connessione, diversa a seconda del tipo di trasporto utilizzato, tra il debugger e la Virtual Machine e prima che avvenga lo scambio di informazioni, effettua un handshake tra i due endpoint. Essendo JDWP *packet-based* e non a stati, lo scambio di informazioni avviene con l'utilizzo di due tipi di pacchetti, i *command packet* e i *reply packet*. I command packet sono inviati dal debugger alla target VM per richiedere informazioni sullo stato del programma o per controllarne l'esecuzione, oppure sono inviati dalla target VM verso il debugger per la notifica degli eventi ritenuti interessanti, ossia degli eventi di cui si sia fatta richiesta di notifica. I reply packet sono invece solo inviati in risposta ai command packet per comunicare il successo o il fallimento di determinate operazioni. Poichè JDWP è asincrono, è possibile inviare più di un command packet senza attendere il corrispondente reply packet.

Data l'implementazione del protocollo, oltre a permettere che il debuggee e il front-end del debugger appartengano a due processi separati e a

implementazioni diverse della VM, oltre che a piattaforme diverse, è anche possibile avere il front-end del debugger in un linguaggio diverso da Java, caratterizzando JDWP come un protocollo estremamente flessibile, dagli ampi utilizzi e portabile su tutte le VM che implementino l'interfaccia JVMDI.

6.1.2 L'interfaccia JVMDI

JVMDI costituisce la parte in-process[1] di JPDA e l'interfaccia più a basso livello su cui si basa JDWP stesso, essendo interamente sviluppata all'interno della stessa VM. In particolare, JVMDI fornisce un modo per controllare l'esecuzione e ispezionare i vari stati di un'applicazione Java, definendo tutti i dettagli specifici che la Virtual Machine può offrire a un debugger, come le informazioni sullo stack, azioni come il posizionamento di un breakpoint o le notifiche della maggior parte degli eventi di cui sia utile avere un resoconto. Il client JVMDI, che decide di quali eventi avere notifica, è in esecuzione sulla stessa VM del processo oggetto del debugging e ha accesso a JVMDI attraverso un'interfaccia nativa e in-process che permette di avere il massimo controllo sul debuggee e di essere meno intrusivo possibile. E' possibile controllare i client JVMDI con un processo separato che implementi le funzioni principali di debugging senza interferire con l'esecuzione dell'applicazione target.

Le funzionalità offerte da JVMDI riguardano principalmente la gestione della memoria, l'esecuzione dei thread, l'accesso allo stack frame[2] e alle variabili locali, il posizionamento di breakpoint, la notifica di eventi e le informazioni sulle classi e i loro campi.

[1] Un debugger *in-process* è un debugger il cui codice esegue nella stessa Virtual Machine del debuggee e dunque il controllo su di essa ha origine nello stesso processo. Il codice di un debugger *out-of-process*, invece, esegue in VM separate e così il controllo del debugger sul debuggee ha origine in un processo diverso.

[2] Lo stack frame indica lo stato dell'invocazione di un metodo sullo stack delle chiamate di un thread. Durante l'esecuzione di un thread, vengono effettuate push e pop degli stack frame dal call stack in corrispondenza all'invocazione dei metodi.

6.1.3 L'interfaccia JDI

Interamente sviluppata in java, JDI è l'interfaccia out-of-process e ad alto livello di JPDA, le cui funzionalità si adattano alla maggior parte dei debugger. JDI fornisce informazioni utili per i tool di debugging che debbano accedere allo stato di una Virtual Machine in esecuzione e potenzialmente remota, offrendo una vista sulle classi, gli array e le interfacce e offrendo inoltre la possibilità di sospendere e riattivare thread, di collocare i breakpoint, di ispezionare lo stato dei thread (se sospesi, in sleep o attivi), il valore delle variabili e la notifica delle eccezioni e del caricamento delle classi. Essendo il livello più alto di JPDA e quello maggiormente rivolto alle esigenze degli utenti, il suo utilizzo libera dal doversi occupare degli aspetti relativi alle specificità delle singole implementazioni delle diverse VM.

Nel paragrafo successivo si analizzano gli aspetti relativi al rapporto tra debugger e Virtual Machine oggetto di debugging e in che modo essi comunicano.

6.1.3.1 Le connessioni tra debugger e target VM

Sebbene non sia specificato alcun meccanismo di trasporto specifico, il concetto di **transport** (vedi [9]) in JPDA è comunque fondamentale per capirne l'architettura. Un transport è il meccanismo di comunicazione tra il debugger e la *target Virtual Machine*, ossia la Virtual Machine dell'applicazione su cui eseguire il debugging. A seconda del tipo di connessione che si stabilisce tra questi due attori, uno dei due agisce da server e aspetta che l'altro, il client, si connetta. In JPDA, sia il debugger che la target VM possono agire da server e aspettare che l'altro stabilisca la connessione, la quale avviene utilizzando gli indirizzi di trasporto (*transport address*), ossia stringhe dal formato diverso a seconda dell'implementazione del transport, che identificano il server a cui connettersi.

E' possibile utilizzare due meccanismi di trasporto delle informazioni: la memoria condivisa, disponibile solo su piattaforma Windows, e i socket.

Il meccanismo di *shared memory transport* utilizza primitive sulla memoria condivisa che permettono di veicolare informazioni tra il debugger e la

target VM che devono però risiedere sulla stessa macchina. Tale meccanismo, identificato dalla stringa `dt_shmem`, si basa su indirizzi formati da stringhe con i quali Windows può effettuare il mapping tra nomi e oggetti.

Il meccanismo che utilizza i *socket*, identificato dalla stringa `dt_socket`, utilizza il canale TCP/IP per le comunicazioni tra il debugger e la Virtual Machine, che può dunque risiedere anche su macchine diverse, a differenza di quanto accade con il meccanismo dello shared memory transport. I transport address con cui si identificano i due endpoint della connessione, ad esempio nel momento in cui un client deve collegarsi con un server, sono della forma `<host>:<port>` dove `<host>` è il nome dell'host a cui connettersi e `<port>` è il numero della porta su cui il server è in ascolto. Nel caso in cui il debugger e la target VM risiedano sulla stessa macchina, il transport address è identificato solo dalla porta.

Al concetto di transport è legato un altro concetto di pari importanza, quello di **connector**. Un connector è un'astrazione JDI necessaria a stabilire una connessione tra debugger e target VM ed è identificato da coppie nome/valore diverse a seconda della specificità di ogni connector. Le varie implementazioni di JDI possono fornire altrettante implementazioni dei connector corrispondenti ai transport utilizzati e alle VM supportate. In generale ad ogni connector è associato più di un meccanismo di trasporto e varie modalità di connessione tra debugger e Virtual Machine (launching, listening e attaching). Da qui derivano i tre principali tipi di connector, descritti di seguito.

1. `LaunchingConnector`: il debugger esegue, ossia 'lancia' la target VM stabilendo in questo modo la connessione. E' lo scenario più comune e anche il più semplice, dal momento che è il connector stesso ad occuparsi dei dettagli dell'invocazione della nuova VM. Il transport utilizzato per stabilire la connessione varia a seconda della piattaforma sottostante: sotto Windows si utilizza lo shared memory transport, sotto Linux e Solaris il socket transport. Ciò che si ottiene dall'invocazione del metodo `com.sun.jdi.LaunchingConnector.launch(java.util.Map)` è un

mirror[3] VirtualMachine, vale a dire una VM su cui è possibile interagire ai fini del debugging. L'oggetto di tipo Map parametro del metodo launch rappresenta le modalità con cui si desidera lanciare la target VM: le opzioni da sottoporre al comando java, eventuali parametri della classe contenente il main dell'applicazione da eseguire e altri per i quali si rimanda alla documentazione Sun.

2. `AttachingConnector`: il debugger si 'attacca' a una VM in esecuzione, la quale, perchè ciò sia possibile, deve essere stata lanciata con specifiche opzioni del tipo
   `-Xrunjdwp:transport=dt_socket,server=y,address=myhost:8000 -Xdebug -Xnoagent`
   Nell'esempio soprastante si specifica che si utilizzerà il socket transport, che la target VM agirà da server in attesa di connessione da parte del debugger e che essa rimarrà in ascolto sulla porta 8000 dell'host myhost (in caso di debugger e target VM sulla stessa macchina address conterrà solo la porta). Anche in questo caso ciò che si ottiene dall'invocazione del metodo `com.sun.jdi.AttachingConnector.attach(java.util.Map)`, con parametri l'host e la porta a cui connettersi, è un mirror di tipo `VirtualMachine`.

3. `ListeningConnector`: il debugger aspetta le connessioni da parte della target VM e agisce così da server in attesa su una porta specificata. Anche in questo caso la target VM deve essere stata lanciata con opportune opzioni per permettere la sospensione dell'esecuzione fino a che non sia iniziato il debugging e per permettere la connessione con il debugger. Mentre è possibile che ciascun debugger possa accettare più di una connessione da parte di diverse VM, una VM può essere sottoposta a una sola ispezione da parte di un'applicazione di debugging. Il metodo `com.sun.jdi.ListeningConnector(java.util.Map)` il cui parametro specificherà la porta su cui il debugger ascolta le connessioni, restituirà un oggetto di tipo VirtualMachine.

[3]Un mirror (`com.sun.jdi.Mirror Interface`) è un proxy utilizzato per analizzare un'entità in una VM separata. Per i dettagli si veda [com.sun.jdi.Mirror api]

JDI è organizzata in quattro package che forniscono tutte le funzionalità necessarie ad avviare un debugging su una target VM:

- `com.sun.jdi`: il package principale che definisce tutti i mirror necessari a indagare sui tipi, sulle classi e sulla stessa Virtual Machine;
- `com.sun.jdi.connect`: il package che definisce le connessioni tra il debugger e la target Virtual Machine;
- `com.sun.jdi.event`: un insieme di classi per definire gli eventi che possono essere monitorati in JDI
- `com.sun.jdi.request`: il package utilizzato per richiedere la notifica degli eventi che si ritengono interessanti.

Per i dettagli e gli approfondimenti si rimanda al sito [8] e, per un esempio di gestione degli eventi e delle richieste, al capitolo 9 in cui nell'illustrare il funzionamento di EXaM, si dà ampio spazio alla parte fondamentale svolta dalla libreria JDI.

6.2 RMI

6.2.1 Concetti di base

Java Remote Method Invocation (RMI) permette di invocare un metodo di un oggetto residente in uno spazio di indirizzi[4] diverso, sito sulla stessa macchina o su macchine diverse rispetto all'oggetto che invoca il metodo [10]. In questo modo è possibile eseguire metodi relativi a classi su sistemi in cui esse non siano mai state installate.;

L'invocazione di metodi remoti coinvolge tre processi:

[4]Per spazio di indirizzi si intende un contesto in cui un indirizzo di memoria è valido.

1. un *Client*, ossia il processo che invoca un metodo di un oggetto remoto;
2. un *Server*, ossia il processo che possiede l'oggetto remoto e quindi un oggetto nello spazio di indirizzi del server;
3. un *Object Registry*, vale a dire un name server che associa i nomi agli oggetti corrispondenti. Dopo che un oggetto è stato registrato in questo name server, le altre applicazioni remote possono interrogare l'Object Registry per accedere all'oggetto per mezzo del suo solo nome.

In Java RMI è possibile utilizzare due tipi di classi:

1. una classe di tipo *Remote*, la cui istanza è detta *oggetto remoto,* che identifica gli oggetti le cui istanze possono essere utilizzate da remoto. Un oggetto di questo tipo può essere riferito entro lo spazio di indirizzi a cui appartiene (e quindi agire come un semplice oggetto locale) o da un altro spazio di indirizzi per mezzo di un *object handle*. Passare come parametro un oggetto remoto equivale a passare una copia dell'object handle da uno spazio di indirizzi all'altro;
2. una classe di tipo *Serializable*, ossia una classe le cui istanze possono essere copiate da uno spazio di indirizzi all'altro. L'istanza di una classe Serializable, detta *oggetto serializzabile*, è un oggetto su cui è possibile effettuare il marshalling. Passare come parametro un oggetto serializzabile equivale a copiare il valore dell'oggetto da uno spazio di indirizzi a un altro. Le classi di tipo Serializable implementano l'interfaccia `java.io.Serializable`. La maggior parte delle classi Java, come la classe `String`, sono serializzabili.

In RMI, tutte le classi passate come parametri delle funzioni remote o che costituiscono i valori da loro restituiti sono serializzabili.

Le classi remote sono costituite da un'interfaccia pubblica che estende l'interfaccia `java.rmi.Remote` e in cui ogni metodo dichiara di lanciare almeno un'eccezione di tipo `java.rmi.RemoteException`. La classe che implementa

questa interfaccia estende la classe `java.rmi.server.UnicastRemoteObject`, in modo che i suoi metodi possano essere invocati in modo remoto.

L'interfaccia dell'oggetto remoto rappresenta l'object handle copiato da uno spazio di indirizzi a un altro. Ciò significa che, se il server deve necessariamente disporre sia dell'interfaccia sia della classe che la implementa, al client che invoca il metodo remoto è sufficiente avere l'object handle di tale classe, ossia la sua interfaccia. Il modo in cui il client ottiene l'implementazione del metodo remoto coinvolge i concetti di *stub* e di *rmiregistry*.

Uno stub è il risultato della compilazione di un file `.class` utilizzando il compilatore `rmic`. L'applicazione di `rmic` al file `Hello.class`, per esempio, produrrà l'oggetto `Hello_Stub.class`. Lo stub, creato dal server e invocato dal client, agisce come proxy affinchè il client possa invocare il metodo remoto la cui implementazione risiede sul server. Lo stub agisce come proxy verso l'implementazione della classe corrispondente solo in congiunzione con l'rmiregistry, ossia l'object registry citato sopra che associa i nomi degli oggetti remoti agli oggetti stessi. In particolare, il nome di un oggetto remoto comprende l'indirizzo Internet della macchina su cui è in esecuzione l'rmiregistry su cui l'oggetto remoto è stato registrato, la porta su cui il registry è in ascolto (solitamente la 1099) e il nome dell'oggetto remoto.

L'avvio dell'rmiregistry deve avvenire sulla macchina in sui si trova il server e solo dopo che siano stati generati gli stub. Dopo l'avvio dell'rmiregistry, è possibile eseguire il codice del server.

La registrazione dell'oggetto remoto `Hello` (con interfaccia `HelloInterface` e unico metodo esportato `say()`) da parte del server RMI, ossia dell'applicazione che esporta oggetti remoti avviene con la chiamata seguente

```
Naming.rebind(''rmi://myhost:1099/'', new Hello());
```

Il client che voglia invocare il metodo remoto dovrà dunque ricercare l'oggetto remoto registrato sull'rmiregistry

```
HelloInterface hello =
(HelloInterface)Naming.lookup(''rmi://myhost:1099/'');
```

e poi invocare i metodi della classe hello secondo il metodo tradizionale, come se disponesse localmente di tale oggetto

```
hello.say();
```

La metodologia RMI coinvolge anche numerosi aspetti della sicurezza in Java, per cui è necessario definire un RMISecurityManager per poter scaricare le classi remote. Per questo aspetto, però, rimandiamo ai numerosi tutorial consultabili in Internet. Nel paragrafo successivo verranno invece dettagliati alcuni aspetti relativi al recupero dello stub nel caso in cui il client e il server RMi siano fisicamente collocati su due macchine diverse.

6.2.2 Utilizzo di codebase e HTTP server in RMI

Il Java launcher, ovvero il comando `java` per eseguire un'applicazione, ha il compito di lanciare una Virtul Machine. La nuova Virtual Machine cerca e carica le classi nel seguente ordine[7]:

1. **Classi Bootstrap**: le classi che compongono la piattaforma Java, incluse le classi in `rt.jar` e altri importanti `.jar` file nella directory `jre/lib`.

2. **Classi Extension**: le classi che utilizzano il meccanismo Java Extension, riunite in file `.jar` e collocate nella directory `jre/lib/ext`.

3. **Classi dell'utente**: le classi definite dall'utente e in generale tutte le classi esterne che non utilizzano il meccanismo di extension, dunque identificate dall'opzione `-classpath` da riga di comando o dalla variabile d'ambiente CLASSPATH.

Esistono però classi utilizzate da un'applicazione Java che non esistono in nessuna delle locazioni precedentemente citate. Il meccanismo del *codebase* permette, a tal scopo, di estendere la ricerca delle classi da caricare oltre

il classpath locale, funzionando come un classpath remoto e caricando classi Java potenzialmente remote in una Java Virtual Machine specificandone l'indirizzo.

Il valore della proprietà `java.rmi.server.codebase` rappresenta uno o più URL da cui caricare le classi. In particolare, nel caso dell'RMI, gli indirizzi specificati da questa proprietà sono utilizzati per scaricare gli stub e tutte le classi da loro riferite. Gli URL possono essere della forma "`file:///`" se si riferiscono al filesystem locale o a un filesystem reso accessibile da protocolli come NFS, ma più spesso riferiscono risorse di rete come HTTP o FTP server, dalle quali è possibile pubblicare e rendere disponibili le classi da scaricare. Specificando il codebase in questo modo, il server RMI mette così a disposizione un oggetto remoto riferibile con un nome univoco precedentemente registrato nell'rmiregistry con il comando `Naming.rebind`.

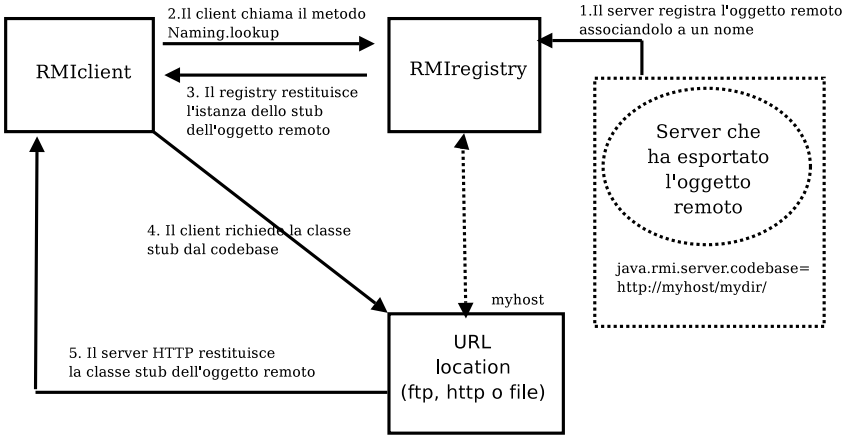


Figura 6.2: Come accedere agli stub RMI [3]

Nel momento in cui un client chiede un riferimento a un metodo remoto chiamando la funzione `Naming.lookup` e interrogando così l'`rmiregistry`, questo restituisce il riferimento remoto, ossia lo stub, cercandolo prima nel classpath locale, poi negli indirizzi specificati nel codebase. In questo modo la definizione dello stub viene scaricata nell'host del client, che può dunque utilizzarla come proxy verso l'oggetto remoto sito sul server. Il client RMI dunque utilizza il codebase per eseguire codice sulla propria JVM. Solitamente il codebase è utilizzato insieme a un HTTP server per scaricare gli

stub remoti: in questo caso l'opzione da passare alla JVM assume la forma

```
-Djava.rmi.server.codebase=http://myhost:2100/
```

se il codebase è stato impostato su un'intera directory contenente vari stub e classi oppure

```
-Djava.rmi.server.codebase=http://myhost:2100/MyClasses.jar
```

se si imposta il codebase su un singolo file jar contenente gli stub e le classi da scaricare.

L'utilizzo di un HTTP server fornisce un meccanismo automatico per scaricare i file stub e permette di avere flessibilità riguardo alla locazione da cui scaricare i file, potendo modificare il modo in cui il downloading viene gestito senza intaccare i server o i client che utilizzano tali classi. Inoltre, poichè le primitive Java sulla sicurezza si basano soprattutto sulla locazione da cui le classi remote arrivano, i permessi relativi alla sicurezza possono specificare l'URL dell' HTTP server da usare congiuntamente con il meccanismo RMI.

6.3 XML e Jakarta Commons Digester

6.3.1 XML

Extensible Markup Language (XML) si è da tempo imposto come standard per il formato dei dati scambiati tra applicazioni distribuite su piattaforme diverse, ossia per tutti quei software che necessitano di comunicare in modo immediato, senza l'appesantimento di ulteriori informazioni di decodifica. La semplicità di XML ha poi permesso che esso venisse usato come formato per la maggior parte dei file di configurazione, promuovendolo così al ruolo di lingua franca dello comunicazione tra software.

XML è un formato testuale immediato e flessibile progettato per descrivere i dati permettendo di concentrarsi esclusivamente su di essi e non sulla

loro rappresentazione. Come HTML, è un markup language in cui però i tag e la struttura del documento non sono predefiniti: ciascun utente può infatti definire i propri a seconda dei dati trattati. In questo modo, mentre HTML visualizza i dati, XML, in modo indipendente, li organizza e permette di scambiarli anche attraverso sistemi diversi, realizzando così una forte indipendenza tra visualizzazione e struttura delle informazioni.

6.3.1.1 Well-formedness e validazione

Data l'ampia libertà nell'organizzazione della struttura di XML, è spesso opportuno definire quali siano i tag ritenuti validi perché un documento XML possa essere letto e correttamente analizzato. A questo proposito, si definisce ben formato (*well-formed*) un file su cui si possa effettuare il parsing e che segua la grammatica XML e una struttura corretta dal punto di vista dell'annidamento dei tag, della loro posizione e della loro chiusura. Si definisce, invece, *valido* un documento che sia conforme alla sintassi e ai vincoli decisi per quel tipo di documento e quindi riporti solo i tag stabiliti dalla document type declaration.

Esistono alcuni strumenti per determinare la correttezza dei file XML a fronte di una descrizione del numero e del tipo di tag obbligatori e opzionali e delle modalità di combinazione. Ad esempio, DTD (*Document Type Definition*) permette di definire gli elementi utilizzabili in un particolare documento XML e la loro struttura, ma alcune limitazioni riguardo i range di validità e le enumerazioni rendono sconveniente utilizzarlo per validazioni complesse. Attualmente più utilizzato di DTD, *W3C XML Schema Definition Language* è un linguaggio che permette di descrivere il contenuto dei documenti XML e controllare se essi rispettino i vincoli specificati, fornendo un meccanismo per la definizione della struttura e della semantica. Esso permette, ad esempio, di specificare quale sia l'elemento `root` del documento, il tipo di ciascun elemento, la sua cardinalità ed eventuali attributi. In questo modo, XML Schema definisce un vocabolario condiviso tra l'ideatore di un documento XML e i suoi utenti, rendendo possibile automatizzarne il controllo della validità e della well-formedness.

6.3.1.2 Acquisizione di documenti XML

Per poter usufruire correttamente dei dati memorizzati in un documento XML, è necessario adottare un metodo per la loro acquisizione e successiva elaborazione. Attualmente, le tecniche più popolari tra quelle disponibili sono il metodo DOM e il metodo SAX.

Il metodo DOM (Document Object Model) consiste nel leggere l'intero documento e crearne una rappresentazione ad albero, fornendo un insieme standard di oggetti per la rappresentazione di documenti XML e HTML e un modello del modo in cui questi oggetti possano essere combinati, insieme a un'interfaccia per accedervi e manipolarli. Semplice da implementare, si dimostra utile per piccoli documenti di cui è necessario acquisire tutti i o quasi i dati contenuti, dal momento che per accedere a un elemento è necessario scorrere tutto l'albero che DOM ha creato e memorizzato.

SAX (Simple API for XML) elabora un documento XML attraverso la risposta agli eventi, rivelandosi appropriato per documenti di grandi dimensioni di cui è necessario accedere solo a una piccola porzione e per i quali DOM si rivelerebbe troppo lento e troppo dispendioso in termini di memoria allocata. SAX, sebbene richieda più impegno da parte del programmatore poichè è meno intuitivo e più elaborato, permette di elaborare i dati al momento del parsing, rivelandosi estremamente utile in caso di accesso veloce e mirato a un documento XML.

6.3.2 Jakarta Commons Digester per l'acquisizione e l'elaborazione di file XML

Il tool open source Digester, proposto da Apache Jakarta, permette di acquisire file XML e di associarli a una gerarchia di oggetti Java [2].

Poiché il metodo DOM di acquisizione è complessivamente più facile da implementare ma più lento e più esoso in fatto di utilizzo di risorse rispetto all'altro, la scelta di Digester è quella di semplificare la tecnica SAX fornendo un'interfaccia ad alto livello verso gli eventi da gestire permettendo così al programmatore di potersi concentrare sui dati XML piuttosto che sulla loro acquisizione e sulla complessità della navigazione del documento.

La classe `org.apache.commons.digester.Digester` richiede che il programmatore specifichi un insieme di azioni da effettuare ogni volta che il parser incontra alcuni pattern nel documento XML. Il framework Digester propone già 10 regole predefinite, come la creazione di un oggetto o il setting di una proprietà, ma è possibile definirne di nuove e implementare tutte quelle necessarie. Ogni volta che Digester incontra uno dei pattern specificati, esso effettua l'azione associata in modo analogo a quello con cui il parser SAX risponde agli eventi legati all'albero XML.

Digester introduce i concetti di *element matching patterns*, di *processing rules* e di *object stack*. Gli *element matching patterns* associano gli elementi in una gerarchia XML alle regole di elaborazione (*processing rules*). Ogni volta che avviene il matching di un pattern, viene eseguita la regola associata. Le *processing rules*, ossia le regole di elaborazione, definiscono ciò che accade quando Digester incontra un pattern. Oltre alle processing rules è possibile anche definire le custom rules, create come sottoclassi di `org.apache.commons.digester.Rule`. Gli oggetti sono resi disponibili dall'*object stack*, per essere manipolati con le processing rules.

Digester permette di trattare i namespaces specificando regole che agiscano solo sugli elementi di un certo namespace ed è anche in grado di effettuare la validazione del file in ingresso con un documento XML-Schema precedentemente specificato.

Per una descrizione dettagliata delle funzionalità di Jakarta Commons Digester si veda [2][12].

Capitolo 7

EXaM - dettagli tecnici

EXaM è stato interamente sviluppato in Java 1.5 in ambiente Linux. Come già accennato in 5.4, è logicamente e strutturalmente costituito da due applicazioni: il master e lo slave. Ciascun slave viene installato su ogni host dedicato all'esecuzione e al monitoring dei singoli componenti dell'applicazione target e comunica con il master, residente su un ulteriore host, per ricevere direttive sul codice da eseguire. In questo capitolo verranno descritti i dettagli tecnici del funzionamento dei componenti master e slave e delle modalità con le quali essi comunicano e si scambiano le tracce delle interazioni tra i processi distribuiti, di cui in Figura 7.1 se ne ha una schematizzazione. I dettagli del monitoring e di come le tracce siano generate sono invece illustrati nel capitolo successivo.

7.1 Il Master

I compiti del componente master di EXaM[1] sono brevemente riassunti nei punti seguenti:

1. configurare l'ambiente per predisporre il monitoring;
2. per ogni esecuzione:

[1]Da qui in avanti si indicherà con *master* il componente master di EXaM e *slave* un singolo componente slave di EXaM

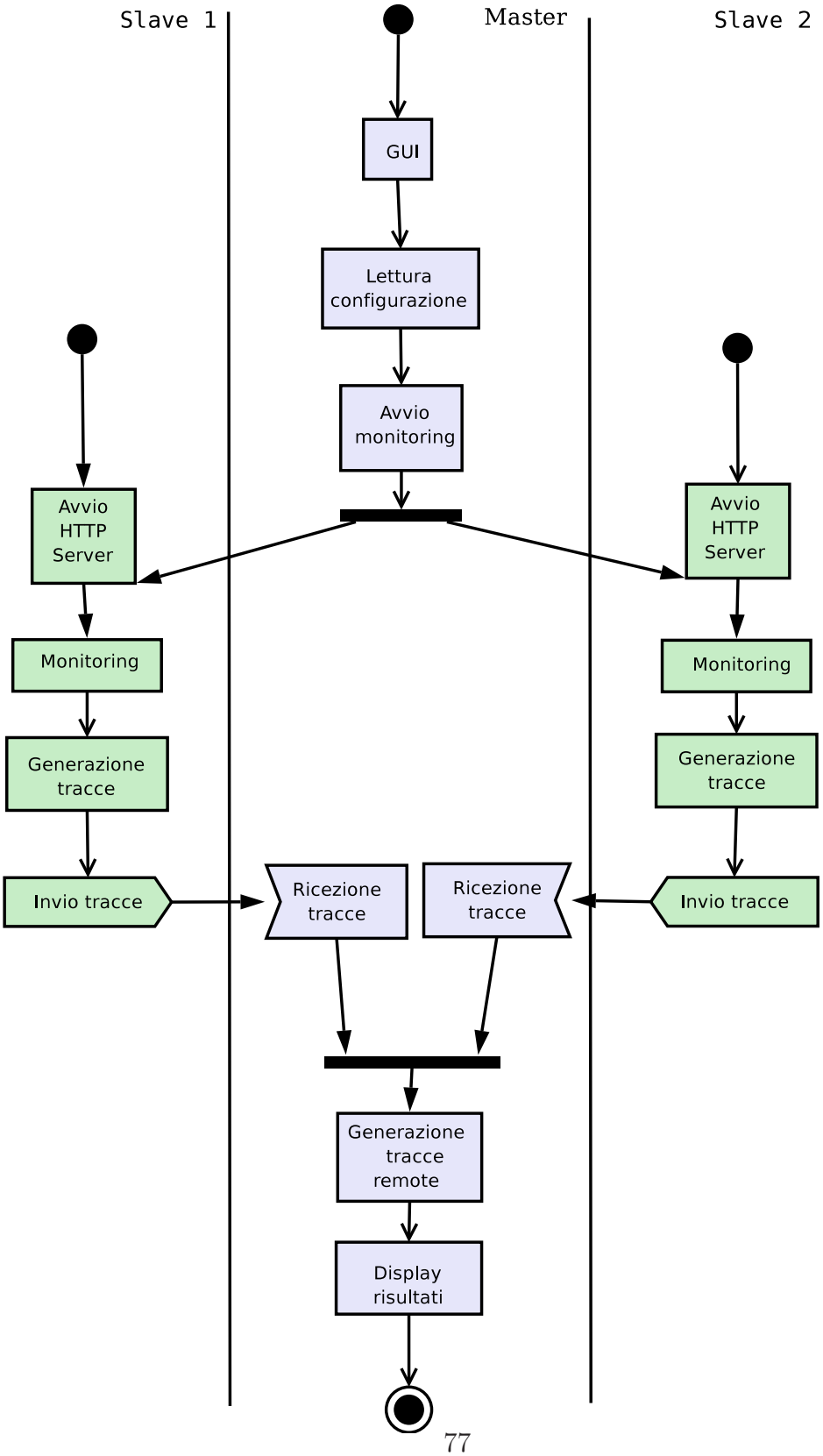


Figura 7.1: Lo schema della comunicazione tra master e slave

(a) leggere le direttive e i dati forniti dall'utente relativi al monitoring;

(b) Per ogni componente coinvolto nel monitoring:

i. avviare il monitoring

ii. scaricare il codice dell'applicazione target

iii. attendere i risultati

(c) generare le tracce riassuntive delle interazioni tra gli oggetti distribuiti.

La lettura delle direttive dell'utente attraverso il file di configurazione è una delle parti più importanti della configurazione del monitoring, poichè è in questa fase che si concretizza il passaggio dalla definizione del sistema in modo astratto per mezzo dell'architettura software alla sua implementazione da parte dei componenti distribuiti sui vari host.

7.1.1 Configurazione

EXaM si occupa di recuperare tutte le informazioni necessarie ad eseguire e monitorare l'applicazione target leggendo il file di configurazione dell'utente. Ciò di cui EXaM necessita per far funzionare il master e gli slave sono i seguenti dati:

- **EXaMIP**: l'indirizzo IP su cui il listener è in esecuzione;
- **exportPort**: la porta attraverso cui è possibile collegarsi al listener;
- **exportDir**: la directory in cui il listener memorizza i file .jar contenenti il codice delle applicazioni target;
- **output**: la directory in cui EXaM può memorizzare i dati ricevuti dai vari slave al termine di ciascun monitoring;
- **host**: un nodo coinvolto nel monitoring. Per ogni nodo su cui verrà eseguito un componente dell'applicazione target viene specificato:

- **name**, **port**: il nome dell'host su cui eseguire il componente e la porta attraverso la quale avverranno le chiamate RMI verso il server (il cui valore, se non altrimenti specificato, è la 1099);
- **code**: il nome del file jar contenente il codice eseguibile per quel componente. Da notare che il nome della classe contenente il main non è necessario per il modo in cui è stato costruito il file jar e per cui si rimanda all'Appendice A;

- **codebase**: un indirizzo di codebase (ad es. http://127.0.0.1:2105/) nel caso in cui questo debba venir specificato come opzione della VM al momento di eseguire il componente. Questo tipo di opzione è piuttosto frequente per applicazioni RMI;
- **tmp**: la directory su cui lo slave può memorizzare il jar e i risultati dell'esecuzione dei componenti;
- **args**: gli eventuali parametri da passare al metodo `main;`
- **policy**: il file di policy per eseguire il componente dell'applicazione target sul nodo in questione, necessario dal momento che il *Security Manager* di Java viene sostituito con un altro affinchè gli scambi RMI possano avvenire. Infatti, ogni volta che viene scaricato codice da un'altra JVM, è necessario proteggere il sistema da operazioni pericolose. E' a questo scopo che la prima operazione in ambito RMI deve essere la creazione di un Security Manager e l'adozione di una politica di sicurezza studiata.
- **package**: la lista dei package del componente da monitorare;
- **interface**: la lista delle interfacce del componente da monitorare.

Il file XML contenente queste informazioni può essere redatto a mano o con l'aiuto di un'interfaccia dello stesso EXaM per guidarne la compilazione, come più avanti illustrato. Per le motivazioni sulla scelta del formato e sul metodo di acquisizione si rimanda al paragrafo 6.3. In Figura 7.2 è riportato un esempio del file XML di impostazioni per EXaM.

```
<tns:EXaM xsi:schemaLocation="http://www.isti.org/EXaM Settings.xsd ">
        <EXaMIP>146.48.84.135</EXaMIP>
        <exportPort>8001</exportPort>
        <exportDir>./jars</exportDir>
        <output>/home/annalisa/workspace/Deployment/Master/output</output>
         <host name="setest0.isti.cnr.it" port="1099">
                <code>Servizi_fat.jar</code>
                <policy>/home/annalisa/Test/RMIServer/conf/java.policy</policy>
                 <codebase>
                        <jar>http://setest0.isti.cnr.it:2299/</jar>
                </codebase>
                <tmp>/home/annalisa/Deployment/tmp/</tmp>
                 <args>
                        <item>1099</item>
                        <item>2299</item>
                        <item>/home/annalisa/Test/RMIServer/jars/</item>
                </args>
                 <package>
                        <item>grossisti.*</item>
                        <item>fornitori.*</item>
                </package>
                 <interface>
                        <item>grossisti.GrossistaInterface</item>
                        <item>fornitori.FornitoriInterface</item>
                </interface>
        </host>
         <host name="146.48.84.135" port="1099">
                <code>Clienti_fat.jar</code>
                <policy>/home/annalisa/workspace/Clienti/conf/java.policy</policy>
                <codebase>
                        <jar>http://setest0.isti.cnr.it:2299/</jar>
                </codebase>
                <tmp>/home/annalisa/workspace/Deployment/Slave/tmp/</tmp>
                <args>
                        <item>setest0.isti.cnr.it</item>
                        <item>1099</item>
                </args>
        </host>
</tns:EXaM>
```

Figura 7.2: Un esempio di file di configurazione per EXaM

Affinchè sia possibile per gli slave scaricare il codice di ciascun componente che costituisce l'applicazione target e, alla fine del monitoring, inviare indietro i risultati, il master avvia un *listener* che si occupa di attendere su una porta specifica le richieste di GET o di SET. In particolare, ogni richiesta di GET è accompagnata dal nome di un file .jar ed è mirata a effettuare il download del file specificato, mentre la richiesta di SET viene effettuata ogni volta che sia necessario trasferire dati al master. Le informazioni relative al file da scaricare per eseguire l'applicazione del componente target e la porta su cui il listener è in attesa sono fornite agli slave come parametri di invocazioni RMI da parte del master, il quale a sua volta le ha acquisite con il parsing del file di configurazione.

Il listener è realizzato come un semplice server a cui i client, ossia gli slave che si connettono, possono collegarsi per trasferire file. Il listener viene attivato con l'avvio del primo monitoring e interrotto con la chiusura di EXaM.

7.1.2 Il monitoring su ciascun componente

Per ogni host indicato nel file di configurazione, EXaM avvia il monitoring chiamando opportune funzioni RMI su ciascun slave. Le informazioni da trasmettere riguardano il file con l'eseguibile target, la porta da cui è possibile scaricarlo e i componenti oggetto del monitoring. Ciascun slave si collegherà poi al listener per ottenere i dati da scaricare. Dopo aver avviato le procedure opportune sugli host coinvolti, EXaM attende che tutti i thread di monitoring ritornino e trasferiscano i dati relativi alle tracce di esecuzione. L'ultima fase del monitoring riguarda la generazione del file XML contenente le interazioni tra i componenti distribuiti, visualizzabile nel *tab* `traces` dell'interfaccia grafica. Il master si limita a raccogliere tutti i file che gli slave hanno provveduto a compilare e a ordinarli in un altro file XML, di cui si parlerà diffusamente nel prossimo capitolo.

7.2 Lo slave

Ciascun slave ha il compito di eseguire il proprio componente e di avviare il monitoring per derivare le tracce di esecuzione, che verranno poi inoltrate alla parte centralizzata di EXaM affinchè un componente esterno ne effettui l'analisi e la validazione.

Come già accennato, l'esecuzione di ciascun componente avviene in modo centralizzato. Si è cercato, cioè, di non presuppore alcuna preinstallazione del componente target sul nodo prescelto ad ospitarlo ma di permettere di guidare tale esecuzione attraverso EXaM. Questa procedura facilita la ripetizione automatica di vari casi di test, dal momento che è sufficiente disporre di una rete più o meno grande in cui sia stato installato, per ogni nodo, la parte slave di EXaM e su cui sia in esecuzione il registro RMI. Il numero e l'identità di nodi impegnati nel monitoring può variare da caso a caso, purché essi appartengano sempre alla stessa rete prescelta. Tale presupposto è necessario per predisporre i permessi (a livello di utenti, di firewall...) necessari per far dialogare il master e gli slave via RMI e per permettere lo scambio di dati. Per quando riguarda l'applicazione target, dunque, si può parlare di *configurazione dinamica e centralizzata*.

7.2.1 L'HTTP Server e la comunicazione RMI

Come già accennato sopra, il master e gli slave comunicano attraverso chiamate RMI, in modo da poter eseguire metodi relativi a classi su sistemi in cui esse non siano state precedentemente installate. Nell'ottica RMI , esiste un'applicazione, detta client, che richiede un metodo di una classe appartenente a un'altra applicazione, il server, di solito sita su un host diverso da quello del chiamante. In EXaM, è il master che si pone come client verso gli slave di cui richiede i metodi necessari ad avviare il monitoring dell'applicazione target. Perchè il master possa accedere allo `stub` delle classi che implementano i metodi richiesti, ciascun slave mette a disposizione un HTTP Server su una porta predefinita che, a ogni richiesta RMI relativa a una delle classi esportate nel rmiregistry, risponde con la spedizione dello

`stub` opportuno. Questo meccanismo funziona poichè è stato specificato, per ogni slave, l'opzione della JVM `-Djava.rmi.server.codebase` che permette di indicare dove recuperare gli stub (vedi paragrafo 6.2.2). Nel caso dello slave, tale opzione di esecuzione della JVM viene impostata con l'indirizzo dell'HTTPServer:

```
-Djava.rmi.server.codebase="http://localhost:2100
```

7.2.2 L'esecuzione e il monitor del componente target

La prima informazione che serve allo slave per poter eseguire l'applicazione target è il nome del file `jar` contenente il codice. Lo slave scarica il `jar` dal listener in esecuzione e invoca il suo metodo `main` leggendolo tra gli attributi del file `MANIFEST`. L'esecuzione e il monitoring avvengono per mezzo delle librerie JDI che ottengono una nuova Virtual Machine dedicata all'esecuzione dell'applicazione target in modalità debugging. Oltre al file .jar, è necessario anche che lo slave conosca eventuali parametri da passare al metodo main e se, oltre all'esecuzione, sia necessario anche il monitoring. In realtà, quest'ultima informazione viene desunta dalla presenza dei nomi dei componenti da monitorare. Se non sono segnalati packages o interfacce di cui tracciare l'esecuzione e se non è indicato alcun punto di breakpoint su cui soffermarsi, lo slave deve dunque limitarsi alla sola esecuzione del componente target, per la quale utilizza comunque le classi fornite da JDI che permettono di invocare una nuova Virtual Machine.

Stabilita dunque la connessione tra il processo che effettua il monitoring e quello che esegue l'applicazione target, è possibile verificare, a livello di Java Virtual Machine, se e quando avvengano eventi ritenuti importanti, come l'ingresso o l'uscita da metodi appartenenti ai packages che si è scelto di analizzare, l'avvio di nuovi thread ma anche l'arresto più o meno improvviso dell'esecuzione. Per ogni evento che si ritiene interessante e per ogni thread avviato viene generato un file XML dal nome univoco e contenente tutte le informazioni necessarie per una futura analisi. Anche le chiamate RMI vengono registrate, ossia tutte le chiamate a metodi remoti che risiedono su host differenti.

E' in questo modo che riusciamo a tracciare le interazioni tra applicazioni distribuite, le quali altrimenti si presenterebbero come *black box* di cui conosciamo solamente gli input e gli output elaborati. Tutti le tracce di esecuzione generate vengono racchiuse in un unico file jar il quale viene poi trasferito al master con una richiesta di SET al listener. Una volta raccolti gli output di tutti i componenti, la parte master di EXaM provvederà a metterli a disposizione di un ulteriore tool preposto all'analisi e alla validazione dei dati.

Lo sviluppo dello slave è avvenuto in modo modulare, permettendo quindi di poter specificare, ed eventualmente sostituire, il tipo di monitoring che si vuole effettuare. A tal scopo è sufficiente provvedere a fornire una classe che implementi le funzioni definite in `MonitorIntf.java`, l'interfaccia implementata da `Monitor.java`, la classe adibita al monitoring. Allo stesso modo, con la separazione della parte di recupero dei dati da quella di analisi, si facilita la scelta e l'aggiornamento della componente che di volta in volta elabora i dati recuperati e segnala eventuali incongruenze rispetto ai requisiti progettuali.

7.3 Interfaccia grafica

L'interfaccia grafica di EXaM è stata interamente sviluppata con la libreria SWT di Eclipse [14][4], una libreria Java nata per permettere agli sviluppatori Eclipse di creare interfacce utente che avessero un look-and-feel nativo e performance proprie del sistema operativo su cui sono in esecuzione. La stessa interfaccia di Eclipse è sviluppata con le SWT.

Il *front-end* di EXaM illustrato in Figura 7.3 è sufficientemente semplice da essere autoesplicativo. Il campo `Setting file` nel *tab* `Settings` permette di specificare il file di configurazione con cui avviare EXaM, selezionabile da *file dialog* attraverso il pulsante `Browse`, mentre i pulsanti `Start`, `Clean` e `Exit` permettono, rispettivamente, di avviare il monitoring, ripulire tutti e tre i tab dai dati dei precedenti monitoring e terminare l'applicazione. Il campo di testo che occupa la maggior parte dell'interfaccia visualizza l'andamento

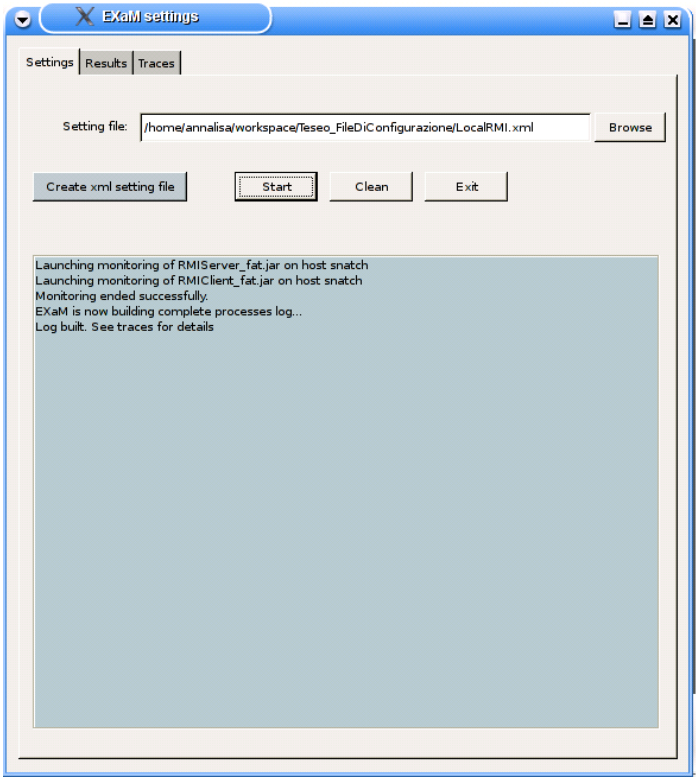


Figura 7.3: Il front-end di EXaM

del monitoring in termini di applicazioni target eseguite. Come già accennato, invece di selezionare un file di configurazione esistente, è possibile crearne uno e salvare le impostazioni di configurazione cliccando sul pulsante `Create xml setting file`. Al clic di questo pulsante, EXaM presenta la schermata illustrata in Figura 7.4, che permette di configurare le impostazioni generiche relative al monitoring - ossia il nome del file che si sta creando, l'indirizzo IP e la porta su cui è in ascolto il listener, la directory di memorizzazione dei jar con il codice delle applicazioni e quella in cui verranno memorizzati i risultati restituiti da ciascun slave. Al termine della compilazione dei primi dati, è possibile accedere alle schermate relative ai componenti target. EXaM ripete questa schermata tante volte quanti sono i componenti da monitorare.

Come mostrato in Figura 7.5, nella schermata relativa alla configurazione di ciascuno degli slave è necessaroio specificare il nome valido di un host nella rete in cui si vuole eseguire un componente target, la porta attraverso cui

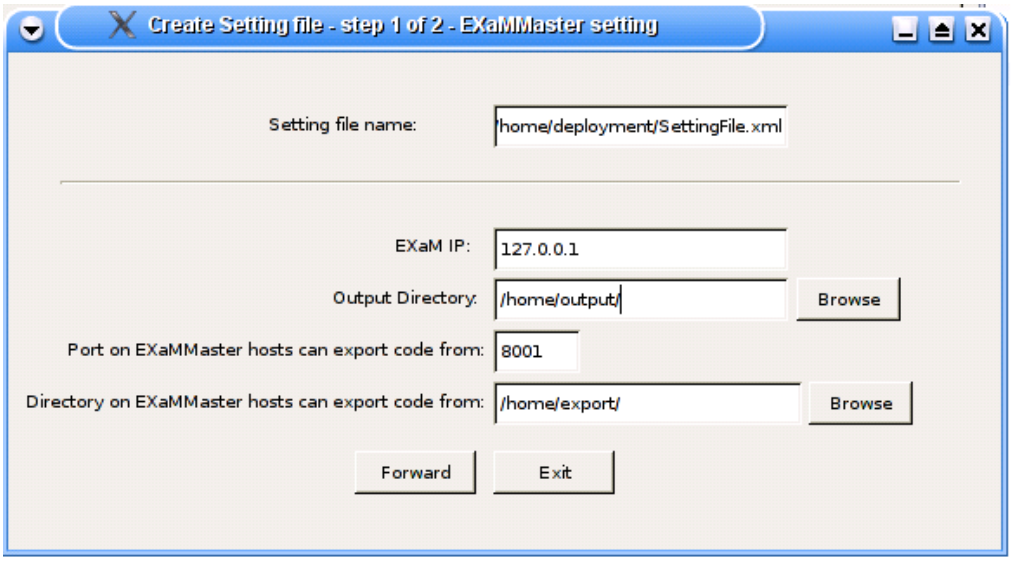


Figura 7.4: La creazione del file di configurazione - step 1

avverranno le chiamate RMI, il nome del file jar contenente il codice dell'applicazione target, l'indirizzo di codebase e il file di policy per poter installare il security manager e scaricare file dal listener. Infine è in questa schermata che vengono specificati i package e le interfacce oggetto del monitoring.

Per ogni monitoring avviato, EXaM visualizza i dettagli e i risultati nel *tab* `Results`. Cliccando sul monitoring di cui interessa sapere i dettagli, come illustrato in Figura 7.6 in cui la riga selezionata appare di colore arancio, EXaM mostra l'output dell'esecuzione del master nel campo `Request` e l'output dell'esecuzione dello slave nel campo `Response`. In realtà, l'output di ciascun componente è costituito dalle scritte a video delle chiamate di sistema, che vengono redirette su un file affinchè siano visualizzabilli a livello di interfaccia grafica. Se l'output del master è disponibile immediatamente, per visualizzare quello dello slave occorre invece attendere che il monitoring sia terminato e che i file con i risultati siano stati trasferiti al master. Se l'utente clicca per conoscere i dettagli del monitoring prima che siano stati trasferiti i dati dello slave, viene riportato un messaggio di non disponibilità dei dati (vedi Figura 7.6). Al termine del monitoring, invece, quando i risultati sono finalmente disponibili, essi sono visualizzabili nel campo `Response`

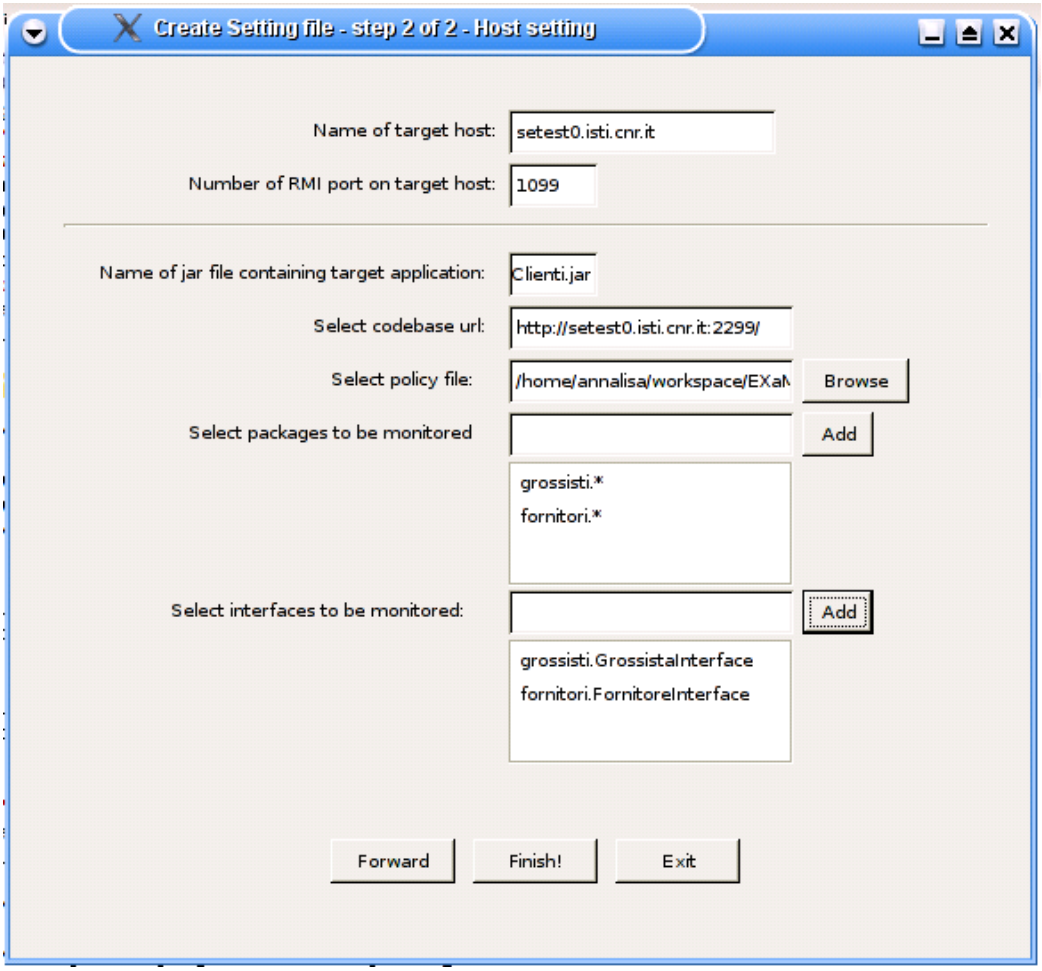


Figura 7.5: La creazione del file di configurazione - step 2

(vedi Figura 7.7). Ovviamente, questi dati sono disponibili anche in formato file e devono essere consultati insieme alle tracce di esecuzione generate come risultato del monitoring.

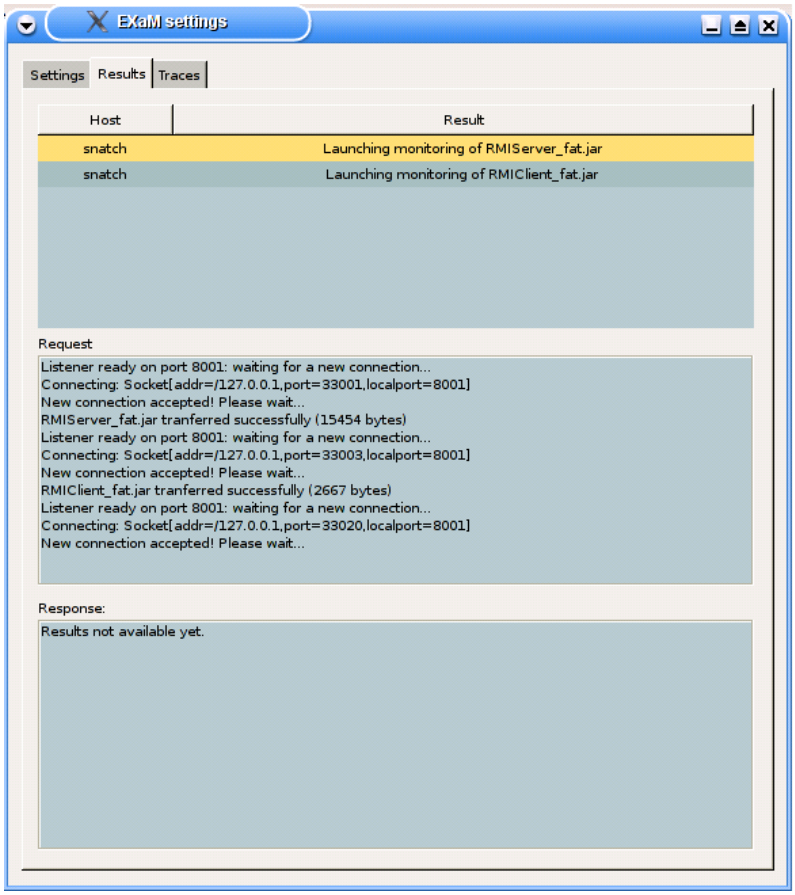


Figura 7.6: Il tab results

Il tab `Traces`, infine, visualizza il file con le interazioni tra i componenti distribuiti come mostra la Figura 7.8. Anche queste informazioni sono disponibili su file XML insieme alle tracce delle interazioni tra i componenti interni, ma la loro visualizzazione nell'interfaccia grafica permette di avere un approccio immediato e molto tangibile sull'andamento del monitoring.

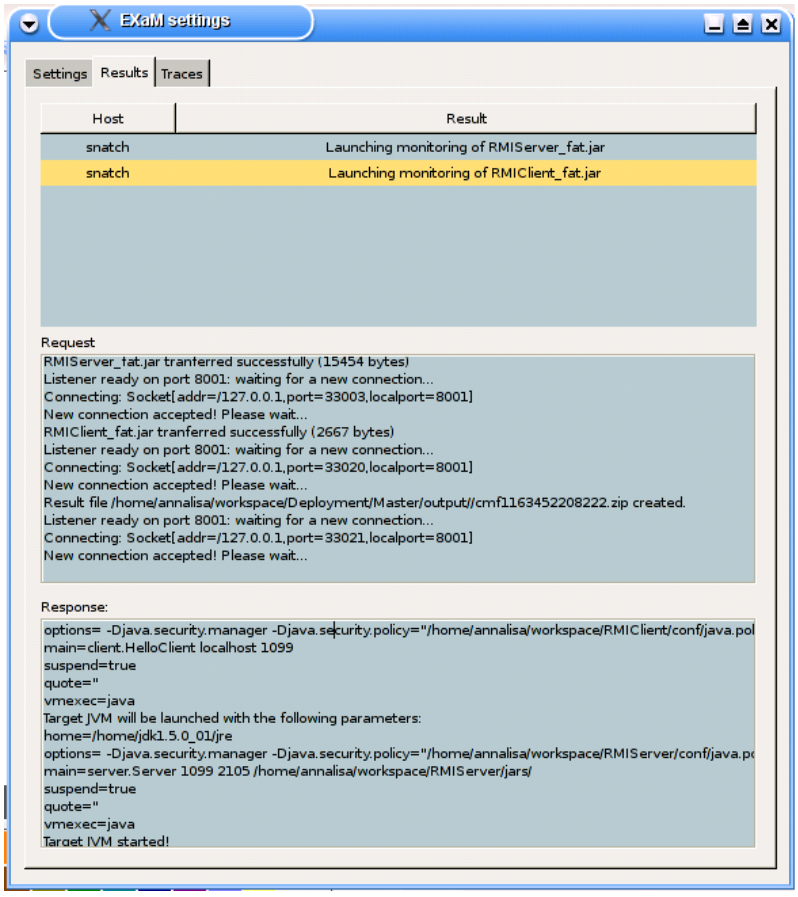


Figura 7.7: Il tab Results alla fine del monitoring

Figura 7.8: Il tab Traces

Capitolo 8

Dettagli della generazione delle tracce

In questo capitolo si forniscono le descrizioni dettagliate del modo in cui l'applicazione target viene lanciata compatibilmente con le regole fornite dalla libreria JDI e in più si definisce in cosa consista il monitoring effettuato da EXaM. Per approfondimenti e rimandi alla libreria utilizzata, si fa riferimento a [8].

Il test di EXaM è stato effettuato su svariate applicazioni, sia distribuite sia locali. Per meglio illustrare che tipo di tracce riesca a generare, si è deciso di riportare quelle ottenute per un'applicazione minimale sviluppata con Java RMI, progettata per evidenziare soprattutto la comunicazione tra componenti diversi.

In Figura 8.1 è rappresentato un package diagram che illustra la struttura del caso di studio, composto da due applicazioni che comunicano attraverso chiamate RMI. La prima, denominata `Servizi`, implementa un semplice sistema di fornitura di servizi attraverso classi che rappresentano commessi, fornitori e grossisti. La seconda, denominata `Clienti`, rappresenta le richieste di servizi realizzate attraverso l'invocazione di metodi su oggetti remoti appartenenti ai package di `Servizi`. Lo scambio di informazioni avviene per mezzo di un'interfaccia condivisa tra `Servizi` e `Clienti`, specializzazione dell'interfaccia `java.io.Serializable`.

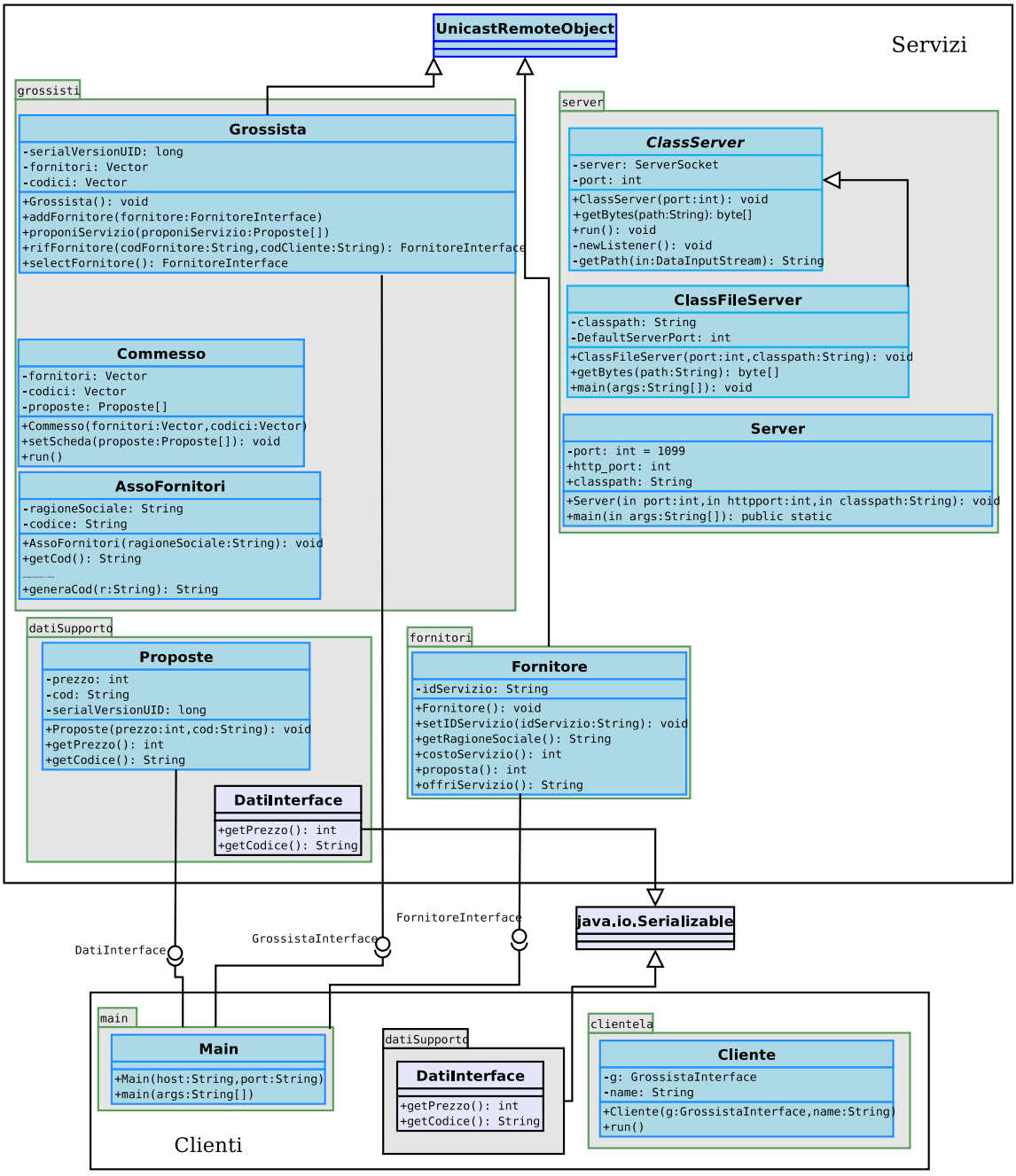


Figura 8.1: Un package diagram del sistema utilizzato per il caso di studio

Ciò che EXaM traccia sono le chiamate dei metodi appartenenti a packages diversi, siano questi locali o remoti. In particolare, dato un file di configurazione come quello mostrato in Figura 7.2, che è poi quello utilizzato per eseguire il monitoring del sistema, saranno visualizzabili le chiamate tra i metodi remoti appartenenti ai packages `grossisti` e `fornitori`, attraverso le interfacce `grossisti.GrossistaInterface` e `fornitori.FornitoreInterface`, così come richiesto nel file di configurazione. Nel seguito di questo capitolo si utilizzeranno i risultati ottenuti dal monitoring su questo file di configurazione per illustrare le tracce ottenute con EXaM e le proprietà architetturali monitorate. Gli estratti di codice riportati sono tutti tratti dal codice di EXaMMaster o EXaMSlave.

8.1 La creazione della nuova Virtual Machine

La prima scelta affrontata è stata quella riguardante il tipo di Connector (vedi 6.1.3.1) che fosse più adatto per la comunicazione tra il debugger e l'applicazione target (anche chiamata debuggee) nel contesto proprio di EXaM.

Era chiaro fin dall'inizio che debugger e debuggee dovessero risiedere sulla stessa macchina per motivi inerenti all'architettura distribuita dell'applicazione da monitorare, ossia perchè il debuggee doveva in primo luogo essere eseguito su un host della rete per ricalcare correttamente un sistema reale, inoltre perchè il master non poteva farsi carico di eseguire il debugger di tutte le componenti distribuite. Non era chiaro invece se l'applicazione target dovesse essere lanciata dal master o da ciascun slave che la monitorava. L'approccio centralizzato che ha guidato tutte le specifiche di EXaM ha portato all'inizio a pensare di avviare l'esecuzione dell'applicazione target per mezzo del master e di permettere allo slave di comunicare con la target VM attraverso un AttachingConnector. L'esigenza successiva di semplificare al massimo la logica di EXaM per favorirne futuri sviluppi e la difficoltà nel lanciare una VM su un host remoto, hanno poi portato alla scelta di trasferire il codice target (compresso in un unico file .jar) sull'host dove risiede lo slave e lasciare a quest'ultimo il compito di eseguirlo per mezzo di un

LaunchingConnector. Tale scelta risolve anche la necessità di dover eseguire comunque il codice target, anche qualora non sia richiesto alcun monitoring sui componenti interni.

L'esecuzione della target VM per mezzo di un LaunchingConnector richiede che siano specificati il metodo `main` da invocare, i parametri da passargli e le opzioni alla Java Virtual Machine, ad esempio riguardo l'utilizzo di codebase o l'adozione di file di policy. Se il parametro `vmexec` non viene specificato, `com.sun.jdi.LaunchingConnector` avvierà l'applicazione target per mezzo dell'eseguibile `java` (alternative prevedono l'utilizzo dei comandi `javaw` o `java_g`).

```
VirtualMachine vm = null;
LaunchingConnector launchConnector =
        (LaunchingConnector) connector;

Map<String, Connector.Argument> launchedarguments =
        connectorArguments(launchConnector);
try {
        vm = launchConnector.launch(launchedarguments);
        return;
} catch (IOException exc) {
        throw
                new Error(''Unable to launch remote target VM: '' + exc);
}
```

Algorithm 1: Esecuzione di LaunchConnector.

In Algoritmo 1 è illustrato il codice necessario al lancio di un LaunchingConnector. `connectorArguments` è una funzione che recupera i parametri da passare al LaunchingConnector (`main` e `options`) e verifica la correttezza di quelli effettivamente passati. Il metodo launch di `com.sun.jdi.LaunchingConnector.launch` restituisce la nuova Virtual Machine di cui si ha completo controllo e a cui si può accedere in modalità debugging.

8.2 La gestione degli eventi

La libreria JDI fornisce la classe `com.sun.jdi.request.EventRequestManager`, di cui ne esiste una sola istanza per ogni Virtual Machine, per catturare gli eventi, come le invocazioni dei metodi, appena questi si verifichino.

Per ogni insieme di invocazioni di metodi da intercettare, si aggiunge un oggetto di tipo `com.sun.jdi.request.MethodEntryRequest` all'Even-

tRequestManager. Nel caso si voglia intercettare solo le classi appartenenti a uno specifico package, come avviene in EXaM in cui si vuole monitorare solo le classi appartenenti ai packages specificati nel file di configurazione, è necessario specificare un *class filter* che restringa gli eventi di cui si richiede la notifica. Nel caso di EXaM, si specifica un class filter per le classi che derivano da `java.lang.Thread` e per le classi appartenenti ai packages da monitorare (ossia quelli specificati nel file di configurazione). Un estratto di codice in cui si richiede la notifica della partenza di ogni nuovo thread e della segnalazione di ingresso dei metodi della classe java.lang.Thread è riportato in Algoritmo 2.

```
ThreadStartRequest thread_started =
        this.events_manager.createThreadStartRequest();
MethodEntryRequest thread_meth_in =
        this.events_manager.createMethodEntryRequest();
thread_meth_in.addClassFilter(''java.lang.Thread'');
thread_meth_in.setSuspendPolicy(EventRequest.SUSPEND_ALL);
thread_meth_in.enable();
thread_started.enable();
```

Algorithm 2: Gestione degli eventi.

L'istruzione `setSuspendPolicy(EventRequest.SUSPEND_ALL)` determina la sospensione dei thread quando si verifica l'evento segnalato nella target VM. Sebbene EXaM, specificando la costante `EventRequest.SUSPEND_ALL`, ottenga la sospensone di tutti i thread, è anche possibile sospendere solo il singolo thread che ha generato l'evento. Si è ritenuto più opportuno sospendere tutti i thread per non alterare le sincronizzazioni esistenti tra i thread e non generare, oltre a un degrado delle prestazioni indotto dalla sospensione, anche squilibri temporali o disallineamenti all'interno dell'esecuzione.

Ogni volta che l'EventRequestManager cattura un evento, l'evento catturato viene posto in cima a una EventQueue. Di solito il corpo principale della gestione degli eventi in un tool di monitoring consiste dunque in un ciclo in cui gli eventi sono estratti dalla EventQueue e da cui si esce solo al verificarsi di una `com.sun.jdi.VMDisconnectedException.`

Ciascun evento viene poi gestito nel modo più appropriato. L'istruzione `eventi.resume()` permette di riavviare la normale esecuzione dei thread, sospesa al verificarsi dell'evento da tracciare. In Algoritmo 3 è riportato un frammento di codice per la gestione di eventi.

```
EventRequestManager events_manager = vm.eventRequestManager();
EventQueue events = vm.eventQueue();
// ...
while (true) {
        EventSet eventi = events.remove();
        Iterator it = eventi.eventIterator();
        while (it.hasNext()) {
                Event curr_event = (Event) it.next();
                switch (getEventCode(curr_event)) {
                        case VMSTARTEVENT:
                        ...;
                        break;
                        case METHODENTRYEVENT:
                        ...;
                        break;
                }
                eventi.resume();
        }
}
```

Algorithm 3: La gestione degli eventi.

8.3 La generazione delle tracce

La politica di sospensione adottata da EXaM è necessaria per accedere allo stack del processo in esecuzione. Lo stack, rappresentato in JDI dalla classe `com.sun.jdi.StackFrame`, rappresenta lo stato dell'invocazione di un metodo sullo stack di chiamate di un thread. Durante l'esecuzione di un thread, ogni volta che si entra o si esce dall'invocazione di un metodo, si esegue rispettivamente la push e la pop degli *stack frame* sul call stack del thread interessato. Il call stack di un thread è realizzato come un `java.util.List` di oggetti di tipo `StackFrame`, ottenibile con l'istruzione `ThreadReference.frames()`. Un'istanza di un oggetto di tipo StackFrame viene creata al momento della sospensione di un thread e distrutta appena il thread viene riattivato ed è per mezzo di questa istanza che si accede alle variabili locali di un metodo e al loro valore. La classe `ThreadReference` rappresenta un accesso diretto allo stato di un thread sulla Virtual Machine: attraverso un `ThreadReference`, è possibile osservare se il thread sia in stato di sospensione, se sia in esecuzione o se sia *zombie*.

In Figura 8.2 nella pagina seguente è rappresentato un sequence diagram che illustra i passi necessari ad accedere allo StackFrame della VM.

E' possibile osservare che, dato un qualsiasi evento, JDI risale al thread che lo ha generato e, da qua, allo StackFrame con le informazioni interessanti. Una volta che si sia ottenuto lo StackFrame, è possibile anche determinare il tipo e la classe di appartenenza dell'oggetto chiamante e di quello chiamato

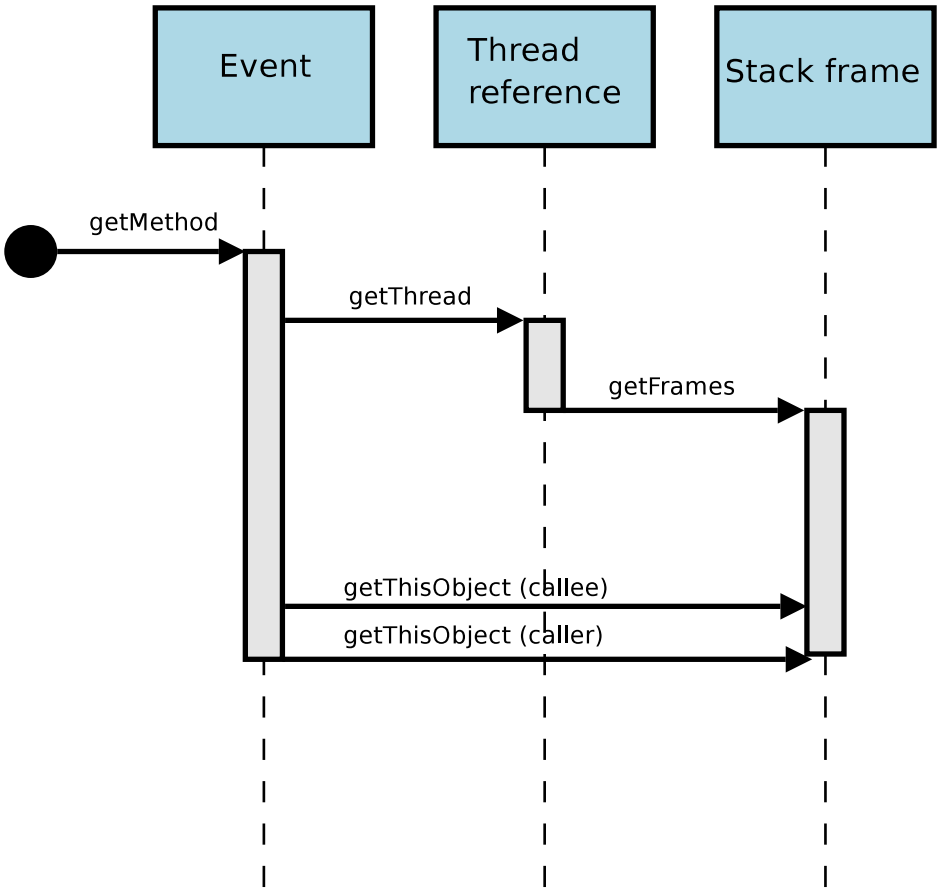


Figura 8.2: Un activity diagram per raffigurare alcune operazioni JDI

e verificare quindi, nell'ottica del monitoring effettuato da EXaM, se essi appartengano allo stesso package o a package diversi.

EXaM genera un file XML per ogni thread avviato, annotandovi ogni ingresso e uscita dai metodi invocati dal thread. Infatti, gli eventi di cui si è fatta richiesta di notifica sono proprio `ThreadStartEvent`, `MethodEntryEvent` e `MethodExitEvent`.

Più dettagliatamente, EXaM chiede la notifica di questi tre eventi, dopodichè, al verificarsi dell'invocazione di un metodo da parte di un thread, crea il file relativo al thread nel caso questo non esista e traccia tutte le informazioni riguardanti la chiamata, quali il nome del metodo, la classe di appartenenza, i valori dei parametri e il tipo dell'oggetto restituito. La

```
<?xml version="1.0" encoding="UTF-8"?>
<thread class="instance of java.lang.Thread(name='main', id=1)" id="1"xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="monitorSchema.xsd">
        <Method entering="yes">
                <methName>java.lang.Thread.start()</methName>
                <thisObject class="java.lang.Thread" id="442"/>
        </Method>
         <Method entering="no">
                <methName>java.lang.Thread.start()</methName>
                 <thisObject class="java.lang.Thread" id="442"/>
        </Method>
         <Method entering="yes">
                <methName>java.lang.Thread.start()</methName>
                <thisObject class="clientela.Cliente" id="774"/>
        </Method>
        <Method entering="no">
                 <methName>java.lang.Thread.start()</methName>
                  <thisObject class="clientela.Cliente" id="774"/>
         </Method>
</thread>
```

Figura 8.3: La traccia relativa al thread 1

procedura di verificare che il file esista, prima della scrittura dei metodi in ingresso e in uscita, è necessaria al fine di evitare di creare file relativi a thread che non invochino neppure un metodo e quindi per limitare la generazione delle tracce e facilitare l'attività successiva di lettura e analisi.

Il file XML rappresentato in Figura 8.3 è relativo alle chiamate in `main.Main` di `Clienti`. E' possibile osservare le invocazioni al thread 442 che si riferisce a una chiamata remota sul package `grossisti` di `Servizi`. Il thread 774 si riferisce invece a un'istanza della classe `clientela.Cliente` create sempre in `main.Main` (in realtà, il file xml riportato in Figura 8.3 è stato semplificato per esigenze di spazio).

Il ritorno dalle invocazioni delle chiamate remote viene tracciato per mezzo della notifica dell'avvio di un thread specifico denominato `DestroyJavaVM`, la cui traccia è raffigurata in Figura 8.4. Un thread di questo tipo è necessario ogni volta che si richiede di liberare le risorse occupate e, nel caso di EXaM, viene avviato al ritorno di una chiamata remota, gestita come una connessione TCP verso l'host che esporta i servizi.

E' per mezzo di questa notazione che EXaM ha notifica del ritorno del thread che ha avviato la chiamata remota e può tracciare l'andamento del-

```
<?xml version="1.0" encoding="UTF-8"?>
<thread class="instance of java.lang.Thread(name='DestroyJavaVM', id=779)" id="779" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="monitorSchema.xsd">
        <Method entering="yes">
                <methName>java.lang.Thread.start()</methName>
                <thisObject class="java.util.logging.LogManager$Cleaner" id="826"/>
        </Method>
        <Method entering="no">
                <methName>java.lang.Thread.start()</methName>
                <thisObject class="java.util.logging.LogManager$Cleaner" id="826"/>
        </Method>
        <Method entering="yes">
                <methName>java.lang.Thread.join()</methName>
                <thisObject class="java.util.logging.LogManager$Cleaner" id="826"/>
        </Method>
        <Method entering="yes">
                <methName>java.lang.Thread.join(long)</methName>
                <thisObject class="java.util.logging.LogManager$Cleaner" id="826"/>
        </Method>
        <Method entering="no">
                <methName>java.lang.Thread.join(long)</methName>
                <thisObject class="java.util.logging.LogManager$Cleaner" id="826"/>
        </Method>
        <Method entering="no">
                <methName>java.lang.Thread.join()</methName>
                <thisObject class="java.util.logging.LogManager$Cleaner" id="826"/>
        </Method>
</thread>
```

Figura 8.4: Il thread 779 corrispondente al ritorno da un'invocazione remota

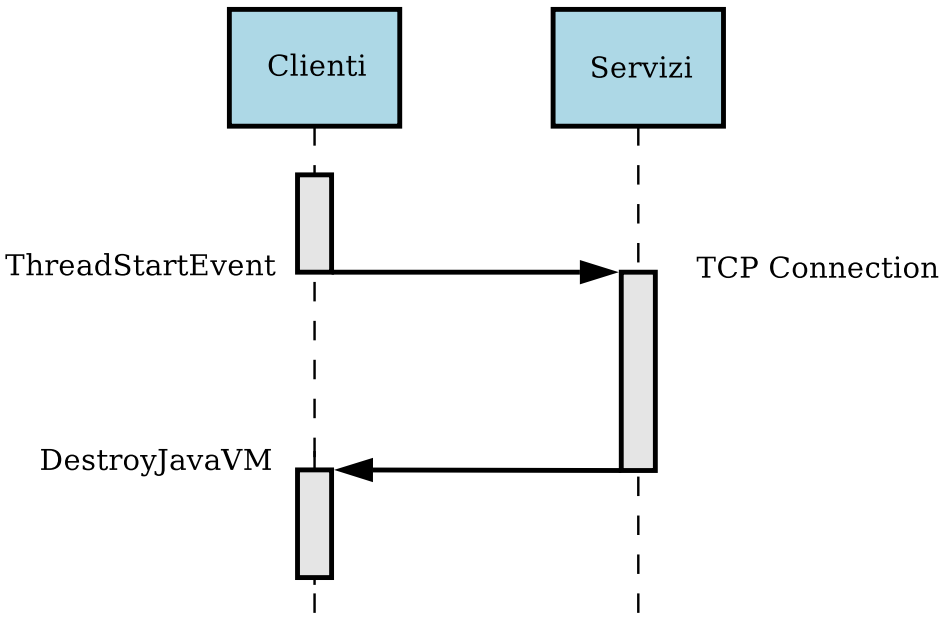


Figura 8.5: Un sequence diagram raffigurante la ricostruzione dell'invocazione delle chiamate remote

le chiamate sul singolo componente, verificando anche le coordinazioni tra i thread, importante dal punto di vista architetturale (vedi Figura 8.5). EXaM delega poi ad altri strumenti l'analisi del rispetto della proprietà della coordinazione tra chiamate, limitandosi a tracciarle nell'ordine in cui esse avvengono fornendo comunque un valido supporto.

In Figura 8.6 osserviamo invece l'avvio, in seguito a richiesta, delle chiamate remote nel package `grossisti` dell'applicazioni `Servizi` e il modo in cui EXaM riesca a rilevare la classe e il metodo invocati e il tipo restituito, anche nel caso non sia un tipo primitivo ma un oggetto serializzabile scambiato tra le due applicazioni remote, come nel caso di `datiSupporto.DatiInterface[]`. Da notare anche il rilevamento dell'host che ha invocato la funzione remota nel nome stesso del thread (`RMI TCP Connection(5) - 146.48.84.135`).

Attraverso l'analisi dei file XML generati da EXaM, è dunque possibile studiare la sequenza di invocazioni tra i thread dei package monitorati, evidenziando proprietà architetturali di coordinazione e sincronizzazione tra le chiamate. In Figura 8.7 è possibile osservare la sequenza di chiamate tra i

```
<thread class="instance of java.lang.Thread(name='RMI TCP Connection(5)-146.48.84.135', id=720)" id="720"
xsi:noNamespaceSchemaLocation="monitorSchema.xsd">
        <Method entering="yes">
                <methName>grossisti.Grossista.proponiServizio()</methName>
                <return>datiSupporto.DatiInterface[]</return>
                <thisObject class="grossisti.Grossista" id="673"/>
        </Method>
        <Method entering="yes">
                <methName>java.lang.Thread.start()</methName>
                <thisObject class="grossisti.Commesso" id="728"/>
        </Method>
        <Method entering="no">
                <methName>java.lang.Thread.start()</methName>
                <thisObject class="grossisti.Commesso" id="728"/>
        </Method>
        <Method entering="yes">
                <methName>java.lang.Thread.join()</methName>
                <thisObject class="grossisti.Commesso" id="728"/>
        </Method>
        <Method entering="yes">
                <methName>java.lang.Thread.join(long)</methName>
                <thisObject class="grossisti.Commesso" id="728"/>
        </Method>
        <Method entering="no">
                <methName>java.lang.Thread.join(long)</methName>
                <thisObject class="grossisti.Commesso" id="728"/>
        </Method>
        <Method entering="no">
                <methName>java.lang.Thread.join()</methName>
                <thisObject class="grossisti.Commesso" id="728"/>
        </Method>
        <Method entering="no">
                <methName>grossisti.Grossista.proponiServizio()</methName>
                <return>datiSupporto.DatiInterface[]</return>
                <thisObject class="grossisti.Grossista" id="673"/>
        </Method>
        <Method entering="yes">
                <methName>grossisti.Grossista.rifFornitore(java.lang.String, java.lang.String)</methName>
                <return>fornitori.FornitoreInterface</return>
                <thisObject class="grossisti.Grossista" id="673"/>
        </Method>
        <Method entering="no">
                <methName>grossisti.Grossista.rifFornitore(java.lang.String, java.lang.String)</methName>
                <return>fornitori.FornitoreInterface</return>
                <thisObject class="grossisti.Grossista" id="673"/>
        </Method>
</thread>
```

Figura 8.6: L'avvio di una chiamata remota in Servizi nella traccia del thread 720

package, ossia tra i componenti: quando un cliente richiede un servizio a un grossista, questi delega la richiesta a un commesso perché si informi sulle proposte di un servizio offerto dai fornitori. Le proposte vengono poi inoltrate ai clienti i quali ne scelgono una e si informano con il grossista sull'identità del fornitore, in modo da rivolgersi direttamente a lui. Tutte le invocazioni che riguardano un cliente avvengono in remoto attraverso Java RMI.

In un altro file, chiamato `threadxxx_values.xml`, in cui xxx è un numero che identifica univocamente il thread, vengono memorizzati i valori di tutte le variabili sullo StackFrame del thread. Si è scelto di visualizzare il valore che le variabili assumono all'ingresso del metodo invocato, ma avremmo anche potuto generare un file analogo all'uscita da ogni metodo invocato. Essendo una tale eventuale modifica piuttosto semplice da realizzare e irrilevante ai fini dello studio presentato, il secondo file con il valore delle variabili in uscita non viene generato, ma è possibile, in una versione più rifinita del tool, parametrizzare questa funzionalità attraverso il file di configurazione.

Infine, EXaM riporta, in un file separato dal nome `package.MainClass.info` (dove `package` è chiaramente il nome del package e `MainClass` il nome della classe contenente la funzione main) lo standard output dell'applicazione, le informazioni sulla Virtual Machine ed eventuali errori nell'esecuzione dell'applicazione target. In Figura 8.8 e in Figura 8.9 riportiamo rispettivamente il contenuto dei file main.Main.info relativo all'esecuzione di Clienti e server.Server relativo all'esecuzione di Servizi.

8.4 La gestione dei breakpoint

E' possibile specificare uno o più punti di brekpoint su cui soffermarsi nel corso del monitoring. EXaM provvede a sospendere l'esecuzione ogni volta che si raggiunga uno di questi punti, ad accedere allo Stack Frame corrente e a memorizzare in un file il valore di tutte le variabili coinvolte nella riga segnalata e visibili nello Stack Frame.

Specificare un punto di breakpoint consiste nel creare un oggetto di tipo `com.sun.jdi.Location` e quindi segnalare una classe (che in Java corrisponde a un file), un metodo e una linea di codice nel file di configurazione:

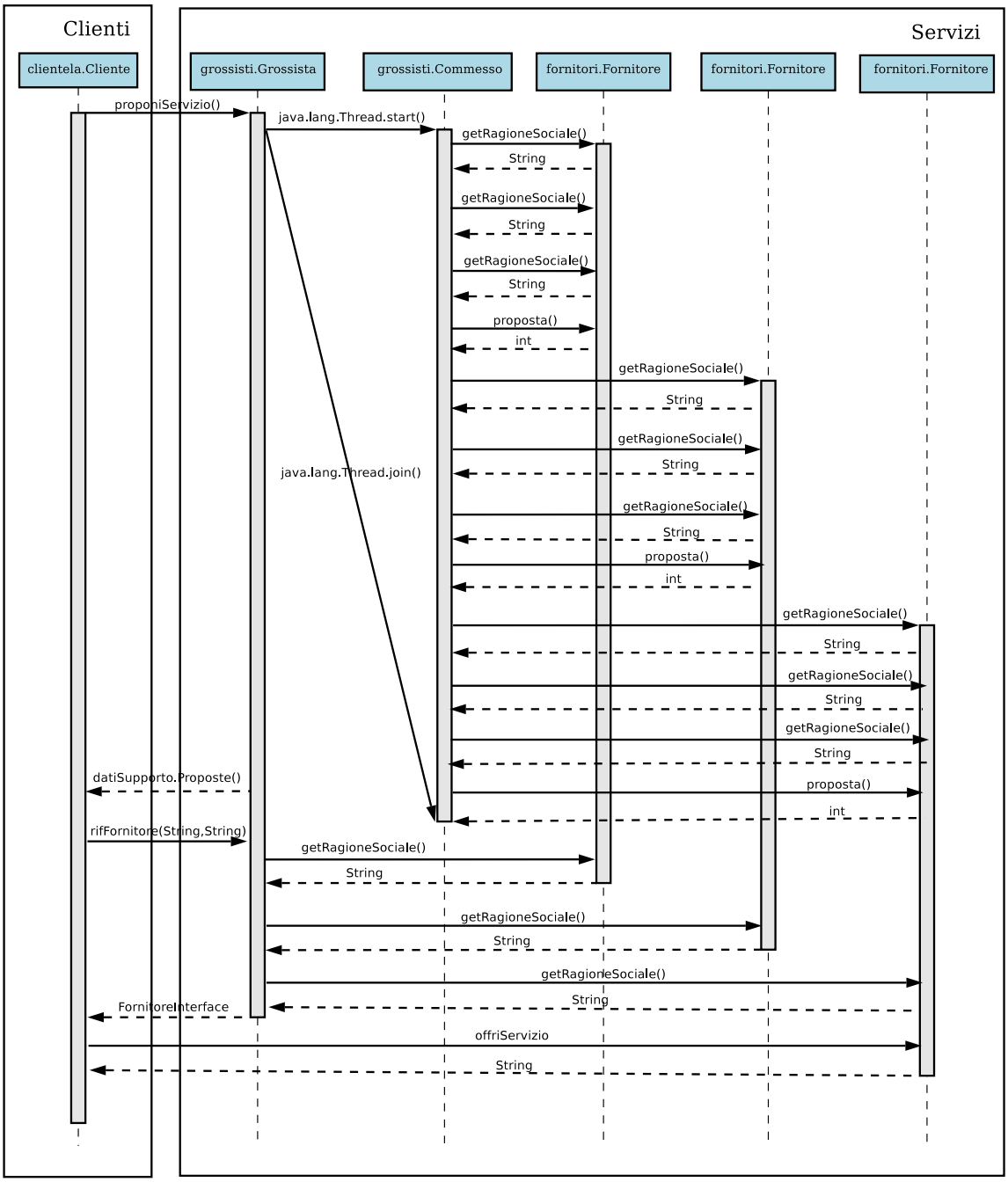


Figura 8.7: La coordinazione tra le invocazioni espressa come sequence diagram

```
Info about main.Main execution:

****************************LAUNCHED JVM INFO*****************************
Target JVM: Java Debug Interface (Reference Implementation) version 1.5
Java Debug Wire Protocol (Reference Implementation) version 1.5
JVM Debug Interface version 1.0
JVM version 1.5.0_01 (Java HotSpot(TM) Client VM, mixed mode)
ClassPath: /home/annalisa/workspace/Deployment/Slave/tmp/Clienti_fat.jar
BaseDirectory: /home/annalisa/workspace/Deployment/Slave
BootClassPath: /home/jdk1.5.0_01/jre/lib/rt.jar
BootClassPath: /home/jdk1.5.0_01/jre/lib/i18n.jar
BootClassPath: /home/jdk1.5.0_01/jre/lib/sunrsasign.jar
BootClassPath: /home/jdk1.5.0_01/jre/lib/jsse.jar
BootClassPath: /home/jdk1.5.0_01/jre/lib/jce.jar
BootClassPath: /home/jdk1.5.0_01/jre/lib/charsets.jar
BootClassPath: /home/jdk1.5.0_01/jre/classes
*************************************************************************
Target application >> Searching for rmi://setest0.isti.cnr.it:1099/Fornitore
Target application >> ...
Target application >> ...
Target application >> ...
Target application >> ...
Target application >> ...
Target application >> ...
Target application >> Fornitore1 recuperato
Target application >> Fornitore2 recuperato
Target application >> Fornitore3 recuperato
Target application >> Searching for rmi://setest0.isti.cnr.it:1099/Grossista
Target application >> Grossista recuperato
Target application >> Fornitore1 aggiunto
Target application >> Fornitore2 aggiunto
Target application >> Fornitore3 aggiunto
Target application >> Il cliente Pietro richiede il servizio a Gioo.
Target application >> Il cliente Luca richiede il servizio a Gioo.
Target application >> Il cliente Gianni richiede il servizio a Gioo.
Target application >> Arrivederci dal fornitore Giovanni & co. al cliente Gianni
Target application >> Arrivederci dal fornitore Giovanni & co. al cliente Luca
Target application >> Arrivederci dal fornitore Giovanni & co. al cliente Pietro

No errors.
```

Figura 8.8: Il contenuto di main.Main.info

```
Info about server.Server execution:

*****************************LAUNCHED JVM INFO*****************************
Target JVM: Java Debug Interface (Reference Implementation) version 1.5
Java Debug Wire Protocol (Reference Implementation) version 1.5
JVM Debug Interface version 1.0
JVM version 1.5.0_06 (Java HotSpot(TM) Client VM, mixed mode)
ClassPath: /home/annalisa/Deployment/tmp/Servizi_fat.jar
BaseDirectory: /home/annalisa/Deployment/Slave
BootClassPath: /usr/java/jdk1.5.0_06/jre/lib/rt.jar
BootClassPath: /usr/java/jdk1.5.0_06/jre/lib/i18n.jar
BootClassPath: /usr/java/jdk1.5.0_06/jre/lib/sunrsasign.jar
BootClassPath: /usr/java/jdk1.5.0_06/jre/lib/jsse.jar
BootClassPath: /usr/java/jdk1.5.0_06/jre/lib/jce.jar
BootClassPath: /usr/java/jdk1.5.0_06/jre/lib/charsets.jar
BootClassPath: /usr/java/jdk1.5.0_06/jre/classes
***************************************************************************
Target application >> HTTP Server installed on http://setest0.isti.cnr.it:2299/home/annalisa/Test/RMIServer/jars/
Target application >> Registering Fornitore...rmi://setest0.isti.cnr.it:1099/Fornitore
Target application >> Fornitore is ready on port 1099!
Target application >> Registering Grossista...rmi://setest0.isti.cnr.it:1099/Grossista
Target application >> Grossista is ready on port 1099!
Target application >> Codice per Giovanni & co.: Gioo.
Target application >> Codice per Giovanni & co.: Gioo.
Target application >> Codice per Giovanni & co.: Gioo.
Target application >> Il commesso <Thread[Thread-11,5,RMI Runtime]> ? attivo alla ricerca di offerte
Target application >> Il commesso <Thread[Thread-12,5,RMI Runtime]> ? attivo alla ricerca di offerte
Target application >> Il commesso <Thread[Thread-10,5,RMI Runtime]> ? attivo alla ricerca di offerte
Target application >> Il commesso <Thread[Thread-11,5,RMI Runtime]> ha terminato la ricerca
Target application >> Il commesso <Thread[Thread-12,5,RMI Runtime]> ha terminato la ricerca
Target application >> Il commesso <Thread[Thread-10,5,RMI Runtime]> ha terminato la ricerca
Target application >> Addebito richiesta codice al cliente: Gianni
Target application >> Addebito richiesta codice al cliente: Pietro
Target application >> Addebito richiesta codice al cliente: Luca

No errors.
```

Figura 8.9: Il contenuto di server.Server.info

```
<breakpoint>
        <class>server.Server</class>
        <method>main</method>
        <line>86</line>
</breakpoint>
```

Figura 8.10: Un esempio di breakpoint

L'evento di raggiungimento di un breakpoint, ossia la creazione di un oggetto di tipo `com.sun.jdi.event.BreakpointEvent`, viene segnalato da JDI appena prima che l'esecuzione arrivi al punto specificato. Quando si raggiunge la linea di codice su cui sia specificata una Location e che quindi soddisfi una richiesta di breakpoint, JDI aggiunge alla coda di eventi della Virtual Machine un event set contenente un'istanza della classe `com.sun.jdi.request.BreakpointRequest`. In questo modo, ottenendo lo StackFrame delle chiamate visibili nella locazione del breakpoint, EXaM riesce ad accedere alle variabili coinvolte e a memorizzarle in un file dal nome `BreakPoint_classname_methodname_line.xml`, che verrà poi compresso insieme alle altre tracce e spedito al master. Nel caso in cui il breakpoint sia posizionato in corrispondenza della chiamata di un metodo, EXaM memorizza anche i parametri attuali e il valore restituito.

8.5 Le interazioni tra gli oggetti distribuiti

Oltre alle interazioni che avvengono internamente a ciascun componente, EXaM traccia anche le interazioni tra due componenti che risiedano su host differenti e che comunichino per mezzo di invocazioni remote di tipo RMI. Anche se la generazione della traccia finale delle interazioni tra i componenti distribuiti spetta al master, ogni slave genera comunque un file XML con le chiamate remote effettuate o ricevute. Quando un componente con un'interfaccia che esporta un metodo remoto (e che dunque estende `java.rmi.Remote`) riceve un'invocazione da un altro componente, la gestione della chiamata RMI viene affidata a un thread dal nome `RMI TCP Connection` (come si è già visto in Figura 8.6). Più propriamente la chiamata RMI tra l'applicazione remota chiamante e il componente chiamato che implementa l'oggetto esportato viene gestita con una connessione TCP, la quale, all'avvio del nuovo thread, viene gestita e tracciata da EXaM.

Ogni slave, dunque, alla fine dell'esecuzione dell'applicazione target, crea un file per ogni interazione remota con il relativo host, dal nome `host1_calls_host2.xml`, in cui memorizza tutte le chiamate remote ricevute leggendole nei file relativi ai thread dal nome TCP-Connection(x)-ip address, come

quello riportato in Figura 8.6). Ciascun file di questo tipo verrà analizzato dal master e combinato per ottenere il file finale con le interazioni remote.

Il master, infatti, attende che ciascun thread ritorni e che tutte le tracce di esecuzione provenienti da ogni host remoto siano state generate. Dopo aver analizzato i file e cercato quelli relativi alle chiamate RMI (ossia i file dal nome `host1_calls_host2.xml`), EXaM Master genera un ulteriore file XML, denominato `RemoteCalls.xml`, in cui tiene traccia di tutte le chiamate remote, ripercorrendo così il flusso di esecuzione durante l'elaborazione tra i componenti. In Figura 8.11 è riportato un estratto di `RemoteCalls.xml`. E' possibile vedere come sia identificabile l'host chiamante e l'host chiamato dagli attributi Caller e Callee.L'ordine in cui i file sono combinati è prettamente sequenziale, dal momento che, non disponendo di tempi comuni tra i vari host della rete, l'identificazione della chiamata che è avvenuta per prima non è un problema di banale soluzione.

```
<?xml version="1.0" encoding="UTF-8"?>
<traces xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="monitorSchema.xsd">
        <Host Caller="146.48.84.135" Callee="setest0.isti.cnr.it">
                <Method entering="yes">
                        <methName>java.lang.Thread.start()</methName>
                        <thisObject class="java.lang.Thread" id="644"/>
                </Method>
                <Method entering="no">
                        <methName>java.lang.Thread.start()</methName>
                        <thisObject class="java.lang.Thread" id="644"/>
                </Method>
                <Method entering="yes">
                        <methName>java.lang.Thread.start()</methName>
                        <thisObject class="java.lang.Thread" id="667"/>
                </Method>
                <Method entering="no">
                        <methName>java.lang.Thread.start()</methName>
                        <thisObject class="java.lang.Thread" id="667"/>
                </Method>
                <Method entering="yes">
                        <methName>grossisti.Grossista.addFornitore(fornitori.FornitoreInterface)</methName>
                        <return>void</return>
                        <thisObject class="grossisti.Grossista" id="673"/>
                </Method>
                <Method entering="no">
                        <methName>grossisti.Grossista.addFornitore(fornitori.FornitoreInterface)</methName>
                        <return>void</return>
                        <thisObject class="grossisti.Grossista" id="673"/>
                </Method>
                <Method entering="yes">
                        <methName>grossisti.Grossista.addFornitore(fornitori.FornitoreInterface)</methName>
                        <return>void</return>
                        <thisObject class="grossisti.Grossista" id="673"/>
                </Method>
                <Method entering="no">
                        <methName>grossisti.Grossista.addFornitore(fornitori.FornitoreInterface)</methName>
                        <return>void</return>
                        <thisObject class="grossisti.Grossista" id="673"/>
                </Method>
                <Method entering="yes">
                        <methName>grossisti.Grossista.addFornitore(fornitori.FornitoreInterface)</methName>
                        <return>void</return>
                        <thisObject class="grossisti.Grossista" id="673"/>
                </Method>
                <Method entering="no">
                        <methName>grossisti.Grossista.addFornitore(fornitori.FornitoreInterface)</methName>
                        <return>void</return>
                        <thisObject class="grossisti.Grossista" id="673"/>
                </Method>
                <Method entering="yes">
                        <methName>grossisti.Grossista.proponiServizio()</methName>
                        <return>datiSupporto.DatiInterface[]</return>
                        <thisObject class="grossisti.Grossista" id="673"/>
                </Method>
                <Method entering="yes">
...
```

Figura 8.11: Il file con le interazioni tra host remoti

Capitolo 9

Conclusioni

Lo studio di una tecnica per il monitoring di applicazioni distribuite ci ha portato ad affrontare varie problematiche relative alla composizione di sistemi a partire da componenti software e alle verifiche più appropriate e più esaustive su architetture di questo tipo. EXaM si propone come punto di partenza per una tecnologia mirata all'analisi di applicazioni distribuite che utilizzano la tecnica RMI di Java e volta ad assecondare le necessità degli utenti in modo flessibile. Infatti, la potenza di EXaM risiede nella possibilità di scelta dei componenti da monitorare, dal momento che a ogni avvio di un monitoring è possibile selezionarne di diversi e modificare il tipo di monitoring stesso con l'introduzione di punti di breakpoint. Inoltre la presenza della rete sottostante diviene totalmente trasparente poichè EXaM considera alla stessa stregua i componenti locali e i componenti remoti, tracciando le interazioni dei primi e dei secondi ed evidenziando solo alla fine le connessioni TCP remote.

Tuttavia, come già abbiamo avuto modo di sottolineare, sono auspicabili molti altri interventi in questa direzione per semplificare ulteriormente l'intervento iniziale e aprire la strada ad utenti sempre meno esperti e meno consapevoli della struttura dell'applicazione da monitorare. Inoltre, il tipo di monitoring da effettuare sull'applicazione può essere reso più penetrante e più descrittivo, coinvolgendo, eventualmente, un'analisi su determinati punti critici. Inoltre, la scelta fatta da EXaM riguarda esclusivamente con-

nessioni remote effettuate tramite Java RMI, ma una versione successiva di questo tool potrebbe prendere in considerazione tecnologie diverse seppur a discapito di un'analisi mirata e specifica.

Riteniamo sia determinante, infine, espandere la funzionalità di EXaM di generare casi di test con il minimo intervento umano, in congiunzione alla prospettiva di rendere sempre più semplice e minimale la configurazione della rete prima di utilizzare questo strumento. Oltre a ciò, si ritiene opportuno integrare in EXaM un linguaggio di specifica più avanzato e più formale, per il quale in questa sede non sono stati effettuati studi sufficientemente analitici e completi.

L'espansione di EXaM in un tool che combini vari aspetti del processo di validazione e una fase finale di testing può decisamente costituire una buona base di partenza per successivi studi che si profilano sempre più numerosi. La necessità di utilizzare strumenti per la verifica di sistemi complessi è infatti sempre più stringente così come stringente e impellente è l'esigenza di riuscire a integrare tool di sviluppo con tool di monitoring e di analisi, in modo da unificare i due processi senza che procedano uno di seguito all'altro, cercando di forgiare quanto più possibile la convinzione che le nuove generazioni di sistemi software complessi debbano procedere di pari passo con strumenti adatti alla loro validazione.

Riconoscimenti

In cima alla lista dei ringraziamenti non può che esserci mia madre che ha sempre creduto in me, qualsiasi scelta facessi. Ringrazio poi mio fratello Andrea, da cui ho imparato la voglia di studiare e approfondire, Daniele Strollo per tutte le volte che mi ha incoraggiato e mi ha fatto forza e Franca Delmastro per il supporto.

Ovviamente i miei ringraziamenti più sinceri vanno poi ad Antonia Bertolino per la possibilità che mi ha dato e ad Andrea Polini per il sostegno costante.

Bibliografia

[1] Charmy project. Technical report. Available at http://www.di.univaq.it/charmy/.

[2] Digester - commons. Technical report. Available at http://jakarta.apache.org/commons/digester.

[3] Dynamic code downloading using rmi. Technical report. Available at http://java.sun.com/j2se/1.4.2/docs/guide/rmi/codebase.html.

[4] Eclipse.org home. Technical report. Available at http://www.eclipse.org/.

[5] Eiffel software. Technical report. Available at http://www.eiffel.com/.

[6] Fujaba tool suite. Technical report. Available at http://wwwcs.uni-paderborn.de/cs/fujaba/projects/reengineering/index.html.

[7] How classes are found. Technical report. Available at http://java.sun.com/j2se/1.5.0/docs/tooldocs/fndingclasses.html.

[8] Java platform debugger architecture. Technical report. Available at http://java.sun.com/j2se/1.5.0/docs/guide/jpda/.

[9] Java platform debugger architecture - jpda connection and invocation. Technical report. Available at http://java.sun.com/products/jpda/doc/conninv.html.

[10] Java rmi tutorial. Technical report. Available at `http://www.ccs.neu.edu/home/kenb/com3337/rmi_tut.html`.

[11] Javis: A uml-based visualization and debugging environment for concurrent java programs. Technical report. Available at `http://uk.builder.com/whitepapers/0,39026692,60093323p-39000971q,00.htm`.

[12] Learning and using jakarta digester. Technical report. Available at `http://www.javaworld.com`.

[13] Plastic. Technical report. Available at `http://www-c.inria.fr:9098/plastic/overview`.

[14] Swt: The standard widget toolkit. Technical report. Available at `http://www.eclipse.org/swt/`.

[15] Welcome to the alamo home page in exile. Technical report. Available at `http://www.cs.nmsu.edu/~jeffery/alamo/`.

[16] The wright architecture description language. Technical report. Available at `http://www.cs.cmu.edu/~able/wright/`.

[17] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[18] A. Bertolino, H. Muccini, and A. Polini. Architectural verification of black-box component-based systems. In N. Guelfi and D. Buchs, editors, *Rapid Integration in Software Engineering, RISE 2006*, pages 1–16, Geneva, 2006. Springer Verlag.

[19] N. Delgado, A. Q. Gates, and S. Roach. An integrated development of a dynamic software-fault monitoring system. Technical report. Available at `www.sdpsnet.org/journals/vol4-3/gates.pdf`.

[20] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Software Eng.*, 30(12):859–872, 2004.

[21] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

[22] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Sept. 2003.

[23] P. Inverardi, H. Muccini, and P. Pelliccione. Charmy: an extensible tool for architectural analysis. In M. Wermelinger and H. Gall, editors, *ESEC/SIGSOFT FSE*, pages 111–114. ACM, 2005.

[24] C. Jeffery. The alamo execution monitor architecture. volume 30, 1999.

[25] P. Mazzoni. Model checking tutorial. Technical report. Available at `http://www.elet.polimi.it/upload/ghezzi/_PRIVATE/DodCheckMazzoni.pdf`.

[26] G. K. Palshikar. An introduction to model checking. Technical report. Available at `http://www.embedded.com/showArticle.jhtml?articleID=17603352`.

[27] S. Rossini and G. Morello. Profiling di applicazioni java. Technical report.

Parte III

Appendici

Appendice A

Deployment

Il deployment di EXaM richiede l'individuazione degli host, in cui i firewall siano stati disabilitati, su cui installare il master e gli slave. E' necessario dunque disporre di una rete di computer su cui siano stati precedentemente installati i tool che costituiscono la parte master e la parte slave di EXaM.

Di seguito si elencano i passi da effettuare per il deployment del master e di ciascun slave come schematizzato in figura A.1.

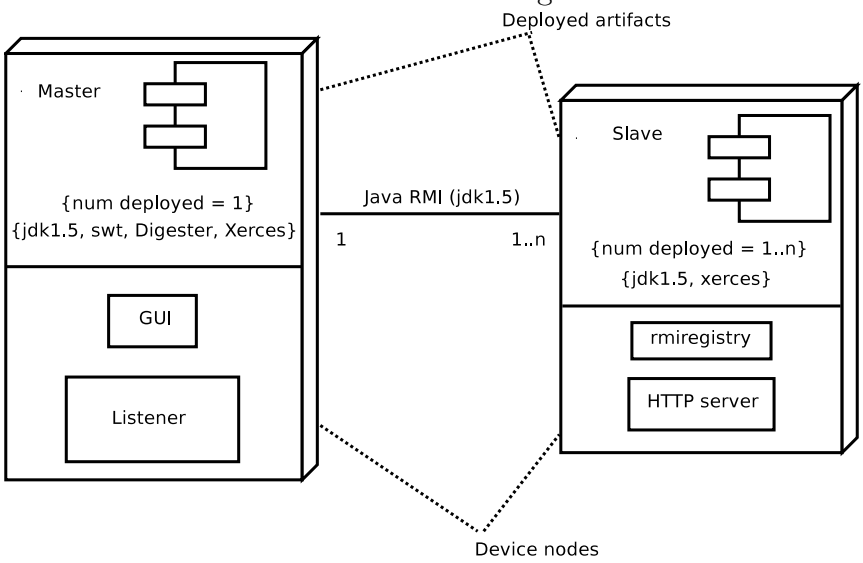


Figura A.1: Il deployment di EXaM

A.1 Deployment del master

Prima di installare il master, è necessario provvedere a soddisfare le seguenti dipendenze:

- SWT di Eclipse Foundation
- Java Runtime Environment 1.5
- Digester di Apache Jakarta
- Xerces per la creazione del file XML di configurazione

Sull'host predisposto a ospitare il server si deve predisporre una directory in cui verranno memorizzati:

- il file `.xml` di configurazione del Master e degli Slave, con le informazioni sulle interfacce e i componenti da monitorare;
- il file *java.policy* con i permessi necessari all'esecuzione dell'applicazione;
- il jar `slave_intf.jar` con l'interfaccia dello Slave `Setup_intf.class`;
- il jar con il codice del `master.jar`;
- i file .jar per Digester;
- i file .jar per SWT;
- la libreria Java `tools.jar`.

- una directory chiamata, ad esempio, `jars`, in cui si provvederà a memorizzare i file `.jar` contenenti il codice dell'applicazione target.
- una directory chiamata, ad esempio, `output`, in cui il listener memorizzerà i file scaricati dagli slave

La classe contenente il main da invocare è `it.cnr.isti.sel.gui.EXaMMaster`.

A.2 Deployment di ciascun slave

Prima di installare ciascun slave, è necessario provvedere a soddisfare le seguenti dipendenze:

- Xerces per la generazione dei file XML
- jre 1.5

Su ciascun host in cui si installerà il lato slave di EXaM si deve predisporre una directory in cui verranno memorizzati:

- il file *java.policy* con i permessi necessari all'esecuzione dell'applicazione;
- i file .jar necessari a Xerces:
  - `/Xerces/xml-apis.jar;`
  - `/Xerces/xercesImpl.jar`;
- la libreria Java `tools.jar`.
- il jar `slave_intf.jar` contenente l'interfaccia `Setup_intf.class`;
- il jar con il codice del `slave.jar`, ossia il file con l'implementazione e gli `stub` dei metodi remoti da invocare con tecnologia RMI, di solito in una directory chiamata `jars` (ad esempio `/home/annalisa/Slave/jars`);
- la libreria tools.jar di Java e le altre librerie necessarie al JDK1.5;

- una directory chiamata, ad esempio, `tmp` per memorizzare i file scaricati dal master e i file con le tracce;
- una directory di nome `jars` in cui memorizzare il file `Setup.jar`,

La classe contenente il main da invocare è
`it.cnr.isti.sel.fileserver.EXaMSlave`
con i seguenti parametri :

- `1099:` la porta per le chiamate RMI;
- `2100:` la porta si cui gira l'HTTPServer necessario a scaricare gli stub e poter così effettivamente invocare i metodi remoti RMI;
- `/home/annalisa/Slave/jars`: la directory in cui l'HTTPServer legge i jar contenenti gli stub.

Prima di invocare la classe suddetta, però, è necessario avviare un rmiregistry. L'rmiregistry deve essere attivo su ogni host che ospita uno slave. Per l'invocazione del `main` è anche necessario specificare il codebase, ossia l'host e la porta da cui sia possibile scaricare gli stub RMI:

1. `rmiregistry &`
2. ```
   java -Djava.security.policy=$(POLICY)
   -Djava.rmi.server.codebase=$(URL)
   -classpath $(CLASSPATH)
   it.cnr.isti.sel.fileserver.EXaMSlave 1099 2100 $(JARDIR)
   ```

dove `$(POLICY)` indica il path del file java.policy, `$(URL)` è l'URL da cui si può accedere l'HTTPServer, `$(CLASSPATH)` è il classpath contenente i jar specificati sopra e `$(JARDIR)` è la directory su cui è attivo l'HTTPServer e dove effettivamente sono memorizzati gli stub.
   ```

A.3 Dettagli sulla costruzione del file jar contenente l'eseguibile dell'applicazione target

La corretta costruzione del file jar contenente l'eseguibile dell'applicazione target è fondamentale per il buon funzionamento di EXaM. Questo file, infatti, serve da classpath al momento di eseguire l'applicazione e di passare i parametri al metodo main e una costruzione errata porterebbe a un errore di tipo `java.lang.ClassNotFoundException` al momento dell'esecuzione del componente target da parte delle librerie JDI. Nel jar è necessario includere tutte le dipendenze necessarie all'applicazione e specificare il nome della classe contenente il metodo main. A questo scopo, durante lo sviluppo e il testing di EXaM si è fatto largo uso del plugin di Eclipse fatjar che risponde pienamente alle esigenze sopra esposte, generando un unico jar contenente tutti i file `.class` richiesti dall'applicazione e compilando in modo appropriato il file MANIFEST con le corrette informazioni sul metodo `main`.

Appendice B

EXaM API Reference

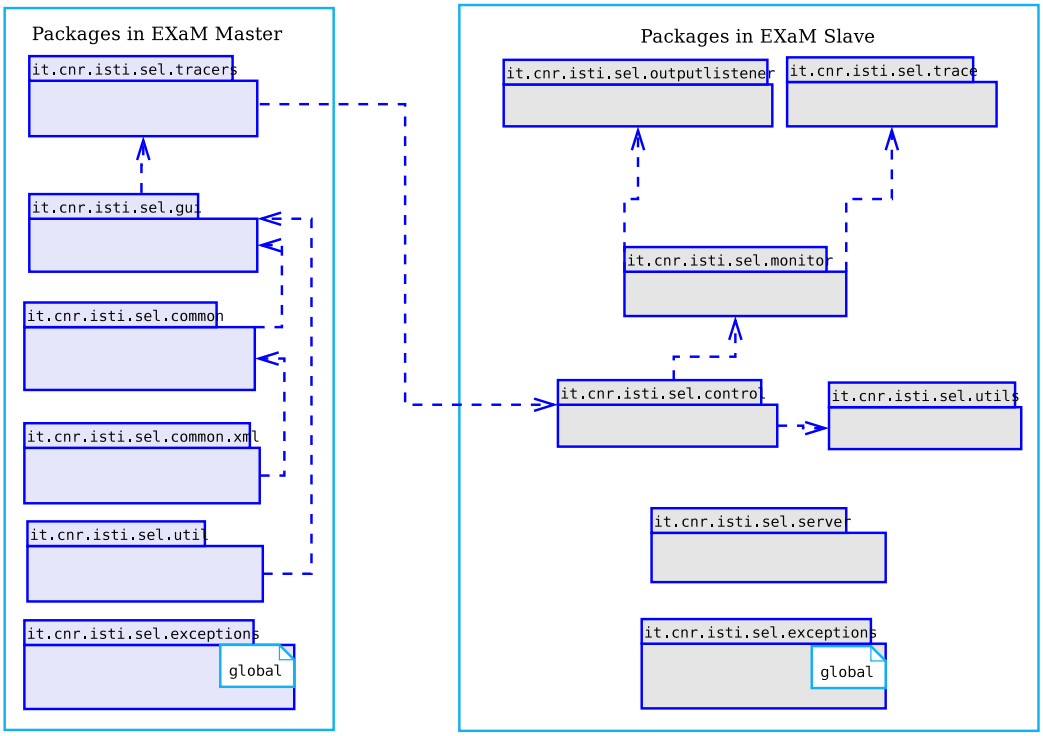


Figura B.1: Un package diagram di EXaM

B.1 EXaM Master Class Documentation

B.2 it.cnr.isti.sel.gui.BrowseButton Class Reference

Public Member Functions

- **BrowseButton** (int x, int y, final Shell shell, final Composite comp, Text target, final boolean onlyDir)

B.2.1 Detailed Description

Class defining a button for browsing file or directory, depending on onlyDir parameter

Author:
Annalisa Bacherotti

B.2.2 Constructor & Destructor Documentation

B.2.2.1 it.cnr.isti.sel.gui.BrowseButton.BrowseButton (int *x*, int *y*, final Shell *shell*, final Composite *comp*, Text *target*, final boolean *onlyDir*)

Class constructor to define a new browse button

Parameters:

x x position of button

y y position of button

shell shell button has to belong to

comp Composite button has to belong to

target button text

onlyDir if true, button browses directory, if false browses files

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/gui/BrowseButton.java

B.3 it.cnr.isti.sel.common.xml.ComponentBuilder Class Reference

Public Member Functions

- **ComponentBuilder** (String xmlSchema)
- **Master getExam** (String filename) throws IOException, SAXParseException, SAXException

Static Public Member Functions

- static void **main** (String[] args)

B.3.1 Detailed Description

This class uses Apache Commons Digester to do the parsing of an xml file after validating it with an XMLSchema file. It does the parsing adding rules to the digester that will be triggered while parsing occurs.

Author:
Annalisa Bacherotti

B.3.2 Constructor & Destructor Documentation

B.3.2.1 it.cnr.isti.sel.common.xml.ComponentBuilder.ComponentBuilder (String *xmlSchema*)

Class constructor

Parameters:
xmlSchema XMLSchema file to validate input file

B.3.3 Member Function Documentation

B.3.3.1 Master it.cnr.isti.sel.common.xml.ComponentBuilder.getExam (String *filename*) throws IOException, SAXParseException, SAXException

Does the parsing of input file firing rules for every tag found.

Parameters:
filename input file to be parsed

Returns:
a **Master**(p. 140) object with ifnormation read from XML input file

Exceptions:
IOException
SAXParseException
SAXException

B.3.3.2 static void it.cnr.isti.sel.common.xml.ComponentBuilder.main (String[] *args*) [static]

Parameters:
args

Exceptions:
ParseException

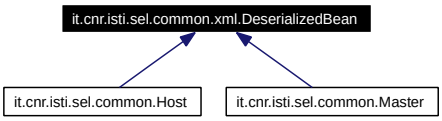
The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/common/xml/ComponentBuilder.java

B.4 it.cnr.isti.sel.common.xml.DeserializedBean Interface Reference

Inherited by **it.cnr.isti.sel.common.Host**, and **it.cnr.isti.sel.common.Master**.

Inheritance diagram for it.cnr.isti.sel.common.xml.DeserializedBean:



B.4.1 Detailed Description

Interface to export validate method

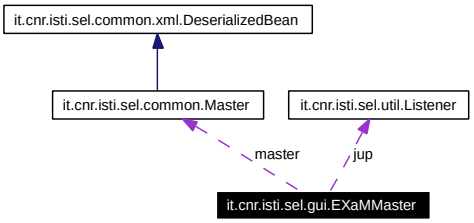
Author:
Annalisa Bacherotti

The documentation for this interface was generated from the following file:

- src/it/cnr/isti/sel/common/xml/DeserializedBean.java

B.5 it.cnr.isti.sel.gui.EXaMMaster Class Reference

Collaboration diagram for it.cnr.isti.sel.gui.EXaMMaster:



Public Member Functions

- **EXaMMaster** ()

Static Public Member Functions

- static void **main** (String[] args)

Package Functions

- **SuppressWarnings** ("static-access") public void startProfiling()

Package Attributes

- Vector< **MonitorInfo** > **hostsInfo**

  vector of MonitorInfo object

B.5.1 Detailed Description

This class, containing main function, displays front-end widgets and loads necessary settings to start execution and monitoring of the target application.

Author:
Annalisa Bacherotti

B.5.2 Constructor & Destructor Documentation

B.5.2.1 it.cnr.isti.sel.gui.EXaMMaster.EXaMMaster ()

Class constructor. Creates widgets for EXaM front-end

B.5.3 Member Function Documentation

B.5.3.1 static void it.cnr.isti.sel.gui.EXaMMaster.main (String[] *args*) [static]

Main function. Creates an instance of EXaMGUI class

Parameters:
args

B.5.3.2 it.cnr.isti.sel.gui.EXaMMaster.SuppressWarnings ("static-access") [package]

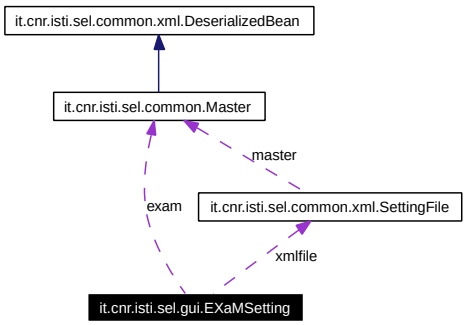
Redirect output on a file, starts monitoring on each host, then creates traces file with remote interactions.

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/gui/EXaMMaster.java

B.6 it.cnr.isti.sel.gui.EXaMSetting Class Reference

Collaboration diagram for it.cnr.isti.sel.gui.EXaMSetting:



Public Member Functions

- **EXaMSetting** ()

B.6.1 Detailed Description

Class to display and save data about EXaM master

Author:
Annalisa Bacherotti

B.6.2 Constructor & Destructor Documentation

B.6.2.1 it.cnr.isti.sel.gui.EXaMSetting.EXaMSetting ()

Class constructor to initialize widgets

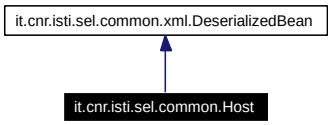
The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/gui/EXaMSetting.java

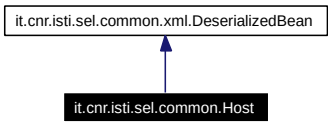
B.7 it.cnr.isti.sel.common.Host Class Reference

Inherits **it.cnr.isti.sel.common.xml.DeserializedBean**.

Inheritance diagram for it.cnr.isti.sel.common.Host:



Collaboration diagram for it.cnr.isti.sel.common.Host:



Public Member Functions

- boolean **validate** () throws ValidationException
- String **getClassName** ()
- String **getMethodName** ()
- int **getLineCode** ()
- void **setClassName** (String bpClass)
- void **setMethodName** (String bpMethod)
- void **setLineCode** (String bpLine)
- String **getCode** ()
- void **setCode** (String code)
- String **getPolicy** ()
- void **setPolicy** (String policy)
- String **getTmpDir** ()
- void **setTmpDir** (String tmpDir)
- Vector< String > **getCodeBaseJars** ()
- void **addCodeBaseJar** (String entry)
- Vector< String > **getInterfaces** ()
- void **addInterface** (String entry)
- Vector< String > **getPackageItems** ()
- void **addPackageItem** (String entry)
- Vector< String > **getArguments** ()
- void **addArgumentsItem** (String entry)
- String **getName** ()
- void **setName** (String **name**)
- int **getPort** ()
- void **setPort** (String **port**)

Public Attributes

- String **name** = ""

  name of target host

- int **port** = 1099

  port for RMI calls (default value 1099)

B.7.1 Detailed Description

Class to store information about each slave. These data are read from XML setting file and parsed by **it.cnr.isti.sel.common.xml.ComponentBuilder**(p. 124) class through Apache Digester

Author:
Annalisa Bacherotti

B.7.2 Member Function Documentation

B.7.2.1 void it.cnr.isti.sel.common.Host.addArgumentsItem (String *entry*)

Adds an entry to arguments vector

Parameters:
entry object to add to arguments vector

B.7.2.2 void it.cnr.isti.sel.common.Host.addCodeBaseJar (String *entry*)

Adds an entry to codebase vector

Parameters:
entry object to add to codebase vector

B.7.2.3 void it.cnr.isti.sel.common.Host.addInterface (String *entry*)

Adds an entry to interfaces vector

Parameters:
entry object to add to interfaces vector

B.7.2.4 void it.cnr.isti.sel.common.Host.addPackageItem (String *entry*)

Adds an entry to packages vector

Parameters:
entry object to add to packages vector

B.7.2.5 Vector<String> it.cnr.isti.sel.common.Host.getArguments ()

Returns:
main arguments vector

B.7.2.6 String it.cnr.isti.sel.common.Host.getClassName ()

Returns:
class name where a breakpoint has been put in

B.7.2.7 String it.cnr.isti.sel.common.Host.getCode ()

Returns:
.jar file name

B.7.2.8 Vector<String> it.cnr.isti.sel.common.Host.getCodeBaseJars ()

Returns:
codebase vactor

B.7.2.9 Vector<String> it.cnr.isti.sel.common.Host.getInterfaces ()

Returns:
interfaces vector

B.7.2.10 int it.cnr.isti.sel.common.Host.getLineCode ()

Returns:
line number where a breakpoint has been put in

B.7.2.11 String it.cnr.isti.sel.common.Host.getMethodName ()

Returns:
method name where a breakpoint has been put in

B.7.2.12 String it.cnr.isti.sel.common.Host.getName ()

Returns:
host name

B.7.2.13 Vector<String> it.cnr.isti.sel.common.Host.getPackageItems ()

Returns:
oackages vector

B.7.2.14 String it.cnr.isti.sel.common.Host.getPolicy ()

Returns:
policy file name

B.7.2.15 int it.cnr.isti.sel.common.Host.getPort ()

Returns:
port number

B.7.2.16 String it.cnr.isti.sel.common.Host.getTmpDir ()

Returns:
directory for temporary files name

B.7.2.17 void it.cnr.isti.sel.common.Host.setClassName (String *bpClass*)

Sets breakpoint class to bpClass

Parameters:
bpClass class name where a breakpoint has to be put

B.7.2.18 void it.cnr.isti.sel.common.Host.setCode (String *code*)

Sets this.code value (pointing to .jar file) to code

Parameters:
code

B.7.2.19 void it.cnr.isti.sel.common.Host.setLineCode (String *bpLine*)

Sets breakpoint line number method to bpLine

Parameters:
bpLine method name where a breakpoint has to be put

B.7.2.20 void it.cnr.isti.sel.common.Host.setMethodName (String *bpMethod*)

Sets breakpoint method to bpMethod

Parameters:
bpMethod method name where a breakpoint has to be put

B.7.2.21 void it.cnr.isti.sel.common.Host.setName (String *name*)

Sets host name to name

Parameters:
name host name

B.7.2.22 void it.cnr.isti.sel.common.Host.setPolicy (String *policy*)

Sets policy file name to policy

Parameters:
policy

B.7.2.23 void it.cnr.isti.sel.common.Host.setPort (String *port*)

Sets port number to port

Parameters:
port number to set port to

B.7.2.24 void it.cnr.isti.sel.common.Host.setTmpDir (String *tmpDir*)

Sets tmpDir name to tmpDir

Parameters:
tmpDir

B.7.2.25 boolean it.cnr.isti.sel.common.Host.validate () throws ValidationException

Validates **Host**(p. 130) with necessary data.

Exceptions:
ValidationException

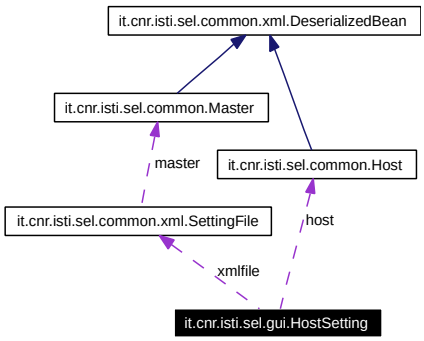
Implements **it.cnr.isti.sel.common.xml.DeserializedBean** (p. 126).

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/common/Host.java

B.8 it.cnr.isti.sel.gui.HostSetting Class Reference

Collaboration diagram for it.cnr.isti.sel.gui.HostSetting:



Public Member Functions

- **HostSetting** (**SettingFile** xmlfile)

B.8.1 Detailed Description

Class to display and save data about each host

Author:
Annalisa Bacherotti

B.8.2 Constructor & Destructor Documentation

B.8.2.1 it.cnr.isti.sel.gui.HostSetting.HostSetting (SettingFile *xmlfile*)

Class constructor

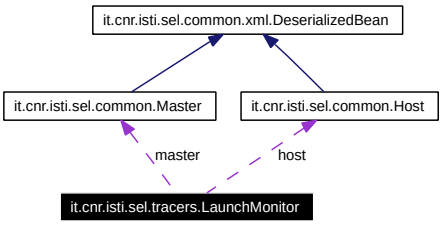
Parameters:
xmlfile xml file with information about EXaM

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/gui/HostSetting.java

B.9 it.cnr.isti.sel.tracers.LaunchMonitor Class Reference

Collaboration diagram for it.cnr.isti.sel.tracers.LaunchMonitor:



Public Member Functions

- **LaunchMonitor** (**Master** master, **Host** host, long ttime) throws RemoteException
- void **start** () throws MonitoringException

B.9.1 Detailed Description

This class launches target VMs and starts monitoring on each target host. Monitoring is started by calling appropriate remote functions on remote hosts through RMI it.cnr.isti.sel.control.SetupIntf interface.

Author:
Annalisa Bacherotti

See also:
SetupIntf

B.9.2 Constructor & Destructor Documentation

B.9.2.1 it.cnr.isti.sel.tracers.LaunchMonitor.LaunchMonitor (Master *master*, Host *host*, long *ttime*) throws RemoteException

LauncheMonitor class constructor

Parameters:
master object with information about master

host name of the host where this target application has to run

ttime name of .dat file where standard output will be redirected

B.9.3 Member Function Documentation

B.9.3.1 void it.cnr.isti.sel.tracers.LaunchMonitor.start () throws MonitoringException

Sets new Security manager, then obtain a remote reference to Setup class to call launchTarget remote method and start target Virtual Machine on remote host.

Exceptions:
MonitoringException

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/tracers/LaunchMonitor.java

B.10 it.cnr.isti.sel.util.Listener Class Reference

Public Member Functions

- **Listener** (String **jarStorage**, int **listeningPort**)
- void **setAlive** (Boolean alive)
- void **setOutputStorage** (String outputStorage)
- void **run** ()

Protected Attributes

- String **jarStorage** = null
  directory where exportable jars are stored
- int **listeningPort** = -1
  port listener is active on

Static Protected Attributes

- static final String **fileSeparator** = System.getProperty("file.separator")
  file separator, different depending on operating system

B.10.1 Detailed Description

This class accepts requests from clients that want to upload or download files from the specified directory on specified ports. Requests are in the form of GET/SET messages.

Author:
Annalisa Bacherotti

See also:
Thread

B.10.2 Constructor & Destructor Documentation

B.10.2.1 it.cnr.isti.sel.util.Listener.Listener (String *jarStorage*, int *listeningPort*)

Class constructor

Parameters:
jarStorage directory where exportable jar files are stores
listeningPort port where requests come

B.10.3 Member Function Documentation

B.10.3.1 void it.cnr.isti.sel.util.Listener.run ()

Thread body. Accepts requests, parses them to see if are SET or GET requets, then calls proper functions.

B.10.3.2 void it.cnr.isti.sel.util.Listener.setAlive (Boolean *alive*)

Set alive variable value. Before EXaMMMaster is closing, it sends a message to the listener to stop through the value of alive variable.

Parameters:
alive variable to set value of boolean class variable alive

See also:
alive

B.10.3.3 void it.cnr.isti.sel.util.Listener.setOutputStorage (String *outputStorage*)

Set outputStorage class variable value

Parameters:
outputStorage

See also:
outputStorage

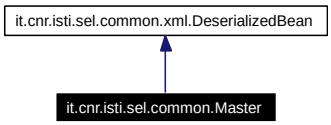
The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/util/Listener.java

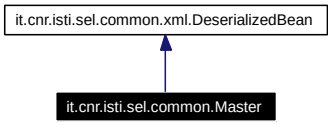
B.11 it.cnr.isti.sel.common.Master Class Reference

Inherits **it.cnr.isti.sel.common.xml.DeserializedBean**.

Inheritance diagram for it.cnr.isti.sel.common.Master:



Collaboration diagram for it.cnr.isti.sel.common.Master:



Public Member Functions

- boolean **validate** () throws ValidationException
- String **getEXaMIP** ()
- void **setEXaMIP** (String **EXaMIP**)
- int **getExportPort** ()
- void **setExportPort** (String **exportPort**)
- String **getExportDir** ()
- void **setExportDir** (String **exportDir**)
- String **getOutput** ()
- void **setOutput** (String **output**)
- Vector< **Host** > **getHosts** ()
- void **addHost** (**Host** host)
- String **getPolicy** ()
- void **setPolicy** (String **policy**)

Package Attributes

- String **output** = ""

  output directory where results will be stored

- String **policy** = ""

  policy file

- String **EXaMIP** = ""

  master IP address

- int **exportPort**

  port .jar files can be exported from (listener port)

- String **exportDir** = ""

directory where .jar files are stored and can be exported to slave (listener directory)

- Vector< **Host** > **hosts** = new Vector<**Host**>()
  slave vector

B.11.1 Detailed Description

Class to store information about master. These data are read from XML setting file and parsed by **it.cnr.isti.sel.common.xml.ComponentBuilder**(p. 124) class through Apache Digester.

Author:
Annalisa Bacherotti

B.11.2 Member Function Documentation

B.11.2.1 void it.cnr.isti.sel.common.Master.addHost (Host *host*)

Adds an host to hosts vector

Parameters:
host host to be added to hosts vector

B.11.2.2 String it.cnr.isti.sel.common.Master.getEXaMIP ()

Returns:
EXaMIP

B.11.2.3 String it.cnr.isti.sel.common.Master.getExportDir ()

Returns:
exportDir

B.11.2.4 int it.cnr.isti.sel.common.Master.getExportPort ()

Returns:
exportPort

B.11.2.5 Vector<Host> it.cnr.isti.sel.common.Master.getHosts ()

Returns:
slave vector

B.11.2.6 String it.cnr.isti.sel.common.Master.getOutput ()

Returns:
output directory

B.11.2.7 String it.cnr.isti.sel.common.Master.getPolicy ()

Returns:
policy file

B.11.2.8 void it.cnr.isti.sel.common.Master.setEXaMIP (String *EXaMIP*)

Sets master IP address

Parameters:
EXaMIP IP address

B.11.2.9 void it.cnr.isti.sel.common.Master.setExportDir (String *exportDir*)

Sets export directory where listener will upload jar files from

Parameters:
exportDir directory name

B.11.2.10 void it.cnr.isti.sel.common.Master.setExportPort (String *exportPort*)

Sets export port where listener will be active

Parameters:
exportPort port number

B.11.2.11 void it.cnr.isti.sel.common.Master.setOutput (String *output*)

Sets output directory

Parameters:
output output directory

B.11.2.12 void it.cnr.isti.sel.common.Master.setPolicy (String *policy*)

Sets policy file to policy

Parameters:
policy name of policy file

B.11.2.13 boolean it.cnr.isti.sel.common.Master.validate () throws ValidationException

Validates master data. If some necessary information is missing, throws an exception

Exceptions:
ValidationException

Implements **it.cnr.isti.sel.common.xml.DeserializedBean** (p. 126).

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/common/Master.java

B.12 it.cnr.isti.sel.common.MonitorInfo Class Reference

Public Member Functions

- **MonitorInfo** (String **hostName**)
- String **getHostName** ()
- void **setOutputFile** (String **outputFile**)
- String **getFile** ()
- void **setResultsFile** (String **resultsFile**)
- String **getResultsFile** ()
- long **getID** ()

Package Attributes

- String **hostName**
  Slave name.
- String **outputFile**
  output file that slave will generate (slave will redirect standard output in this file)
- String **resultsFile**
  results file that slave will generate
- long **id**
  unique monitoring id

B.12.1 Detailed Description

This class is necessary to store information about each host, so that every time a monitoring gets started ExaM master traces slave name, output file slave will generate, results file slave will generate and monitoring id

Author:
Annalisa Bacherotti

B.12.2 Constructor & Destructor Documentation

B.12.2.1 it.cnr.isti.sel.common.MonitorInfo.MonitorInfo (String *hostName*)

Class constructor

Parameters:
hostName host monitoring will get started on

B.12.3 Member Function Documentation

B.12.3.1 String it.cnr.isti.sel.common.MonitorInfo.getFile ()

Returns:
outputFile

B.12.3.2 String it.cnr.isti.sel.common.MonitorInfo.getHostName ()

Returns:
hostname

B.12.3.3 long it.cnr.isti.sel.common.MonitorInfo.getID ()

Returns:
id

B.12.3.4 String it.cnr.isti.sel.common.MonitorInfo.getResultsFile ()

Returns:
resultsFile

B.12.3.5 void it.cnr.isti.sel.common.MonitorInfo.setOutputFile (String *outputFile*)

Sets outputFile

Parameters:
outputFile

B.12.3.6 void it.cnr.isti.sel.common.MonitorInfo.setResultsFile (String *resultsFile*)

Sets resultsFile

Parameters:
resultsFile

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/common/MonitorInfo.java

B.13 it.cnr.isti.sel.exceptions.MonitoringException Class Reference

Public Member Functions

- **MonitoringException** ()
- **MonitoringException** (String msg)

B.13.1 Detailed Description

Class to collect exceptions about monitoring

Author:
Annalisa Bacherotti

B.13.2 Constructor & Destructor Documentation

B.13.2.1 it.cnr.isti.sel.exceptions.MonitoringException.MonitoringException ()

Empty class constructor

B.13.2.2 it.cnr.isti.sel.exceptions.MonitoringException.MonitoringException (String *msg*)

Class constructor with a message that has to be displayed.

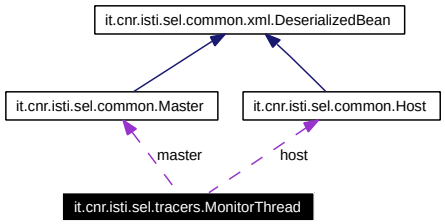
Parameters:
msg

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/exceptions/MonitoringException.java

B.14 it.cnr.isti.sel.tracers.MonitorThread Class Reference

Collaboration diagram for it.cnr.isti.sel.tracers.MonitorThread:



Public Member Functions

- **MonitorThread** (long ttimeID, **Host** host, **Master** master)
- void **run** ()

B.14.1 Detailed Description

Class to start thread on each host as a separate thread

Author:
annalisa

See also:
Thread

B.14.2 Constructor & Destructor Documentation

B.14.2.1 it.cnr.isti.sel.tracers.MonitorThread.MonitorThread (long *ttimeID*, Host *host*, Master *master*)

Class constructor. Stores host to start monitoring in and long ttimeID to associate it with new monitoring.

Parameters:
i host index

host host where monitoring ha sto be started

B.14.3 Member Function Documentation

B.14.3.1 void it.cnr.isti.sel.tracers.MonitorThread.run ()

Thread body. Starts monitoring on host host. Target host also nedds information about master host and ttimeID to give a unique and known name to zip file with monitoring results.

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/tracers/MonitorThread.java

B.15 it.cnr.isti.sel.common.RemoteCall Class Reference

Public Member Functions

- **RemoteCall** ()
- **RemoteCall** (String caller, String callee, long ttime, String ffile)
- String **getCaller** ()
- void **setCaller** (String caller)
- String **getCallee** ()
- void **setCallee** (String callee)
- long **getTime** ()
- void **setTime** (long ttime)
- String **getFile** ()
- void **setFile** (String ffile)

B.15.1 Detailed Description

This class stores information about remote calls for further generation of traces file: caller, callee, file with remote traces and ttime

Author:
Annalisa Bacherotti

B.15.2 Constructor & Destructor Documentation

B.15.2.1 it.cnr.isti.sel.common.RemoteCall.RemoteCall ()

Default class constructor

B.15.2.2 it.cnr.isti.sel.common.RemoteCall.RemoteCall (String *caller*, String *callee*, long *ttime*, String *ffile*)

Clas constructor

Parameters:

caller who started remote call

callee who received remote call

ttime ttime in milliseconds remote call was originated

ffile file with remote interaction

B.15.3 Member Function Documentation

B.15.3.1 String it.cnr.isti.sel.common.RemoteCall.getCallee ()

Returns:
callee

B.15.3.2 String it.cnr.isti.sel.common.RemoteCall.getCaller ()

Returns:
caller

B.15.3.3 String it.cnr.isti.sel.common.RemoteCall.getFile ()

Returns:
ffile

B.15.3.4 long it.cnr.isti.sel.common.RemoteCall.getTime ()

Returns:
ttime

B.15.3.5 void it.cnr.isti.sel.common.RemoteCall.setCallee (String *callee*)

Sets callee

Parameters:
callee

B.15.3.6 void it.cnr.isti.sel.common.RemoteCall.setCaller (String *caller*)

Sets caller

Parameters:
caller

B.15.3.7 void it.cnr.isti.sel.common.RemoteCall.setFile (String *ffile*)

Sets file

Parameters:
ffile

B.15.3.8 void it.cnr.isti.sel.common.RemoteCall.setTime (long *ttime*)

Sets time

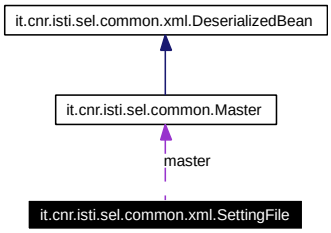
Parameters:
ttime

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/common/RemoteCall.java

B.16 it.cnr.isti.sel.common.xml.SettingFile Class Reference

Collaboration diagram for it.cnr.isti.sel.common.xml.SettingFile:



Public Member Functions

- String **getXMLName** ()
- void **setXMLName** (String xmlName)
- **Master getMaster** ()
- void **setMaster** (**Master** master)
- Vector< **Host** > **getHostList** ()
- void **setHostList** (Vector< **Host** > hosts)
- void **addHost2List** (**Host** host)

B.16.1 Detailed Description

Thia class fills fields to write a new XML Setting file

Author:
Annalisa Bacherotti

B.16.2 Member Function Documentation

B.16.2.1 void it.cnr.isti.sel.common.xml.SettingFile.addHost2List (Host *host*)

Add a host to hostslist

Parameters:
host host to be added to hostslist

B.16.2.2 Vector<Host> it.cnr.isti.sel.common.xml.SettingFile.getHostList ()

Returns:
hostlist

B.16.2.3 Master it.cnr.isti.sel.common.xml.SettingFile.getMaster ()

Returns:
master

B.16.2.4 String it.cnr.isti.sel.common.xml.SettingFile.getXMLName ()

Returns:
xmlName

B.16.2.5 void it.cnr.isti.sel.common.xml.SettingFile.setHostList (Vector< Host > *hosts*)

Sets hostslist

Parameters:
hosts

B.16.2.6 void it.cnr.isti.sel.common.xml.SettingFile.setMaster (Master *master*)

Sets master

Parameters:
master

B.16.2.7 void it.cnr.isti.sel.common.xml.SettingFile.setXMLName (String *xmlName*)

Sets xmlName

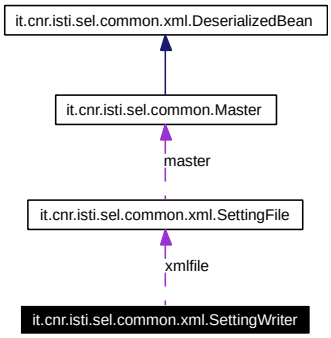
Parameters:
xmlName XML file name

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/common/xml/SettingFile.java

B.17 it.cnr.isti.sel.common.xml.SettingWriter Class Reference

Collaboration diagram for it.cnr.isti.sel.common.xml.SettingWriter:



Public Member Functions

- **SettingWriter** (**SettingFile** xmlfile)

B.17.1 Detailed Description

SettingWirter writes a new XML file readinf data from a **SettingFile**(p. 150) object

Author:
Annalisa Bacherotti

B.17.2 Constructor & Destructor Documentation

B.17.2.1 it.cnr.isti.sel.common.xml.SettingWriter.SettingWriter (SettingFile *xmlfile*)

Class constructor

Parameters:
xmlfile object with information to be written to XML new file

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/common/xml/SettingWriter.java

B.18 it.cnr.isti.sel.tracers.Tracing Class Reference

Public Member Functions

- **Tracing** (String dir, String filename, Vector< **MonitorInfo** > hostsInfo)

B.18.1 Detailed Description

This class reads every zip file in EXaM output directory, then searches in it for files with traces about remote interactions generated by each host and creates another file with every remote interaction

Author:
Annalisa Bacherotti

B.18.2 Constructor & Destructor Documentation

B.18.2.1 it.cnr.isti.sel.tracers.Tracing.Tracing (String *dir*, String *filename*, Vector< MonitorInfo > *hostsInfo*)

Class constructor: fills a list with elements of type RemoteCalls for each traces file, then orders it depending on a time field and creates remote traces file.

Parameters:
dir Directory to search traces file in

filename name of new file with every remote trace from each host

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/tracers/Tracing.java

B.19 it.cnr.isti.sel.exceptions.ValidationException Class Reference

Public Member Functions

- **ValidationException** (String msg)

B.19.1 Detailed Description

Exception thrown while validating XML setting file, if some necessary information is missing

Author:
Annalisa Bacherotti

B.19.2 Constructor & Destructor Documentation

B.19.2.1 it.cnr.isti.sel.exceptions.ValidationException.ValidationException (String *msg*)

Class constructor

Parameters:
msg message shown every time a **ValidationException**(p. 154) is shown

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/exceptions/ValidationException.java

B.20 it.cnr.isti.sel.util.VectorSort Class Reference

Public Member Functions

- **VectorSort** (Vector< **RemoteCall** > vector)
- void **doSorting** ()
- Vector< **RemoteCall** > **getSortedVector** ()

B.20.1 Detailed Description

Class to sort vector of Remote Calls. The sorting algorithm used is quicksort, a significantly faster sorting algorithm based on comparison.

Author:
Annalisa Bacherotti

B.20.2 Constructor & Destructor Documentation

B.20.2.1 it.cnr.isti.sel.util.VectorSort.VectorSort (Vector< RemoteCall > *vector*)

Class constructor

Parameters:
vector the vector to be sorted

B.20.3 Member Function Documentation

B.20.3.1 void it.cnr.isti.sel.util.VectorSort.doSorting ()

Does vector sorting

B.20.3.2 Vector<RemoteCall> it.cnr.isti.sel.util.VectorSort.getSortedVector ()

Returns:
sorted vector

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/util/VectorSort.java

B.21 EXaM Slave Class Documentation

B.22 it.cnr.isti.sel.monitor.BreakpointManager Class Reference

Static Public Member Functions

- static Location **findLocation** (ReferenceType ref, String methodname, int line)

B.22.1 Detailed Description

This class manages breakpoint users decided to place in target code. It gives support in finding Method instances by name and reference a Location instance buy method name and code line number

Author:
Annalisa Bacherotti

B.22.2 Member Function Documentation

B.22.2.1 static Location it.cnr.isti.sel.monitor.BreakpointManager.findLocation (ReferenceType *ref*, String *methodname*, int *line*) [static]

Given a ReferenceType pointing to the class we have to put a breakpoint in, given a method and a line number in target code, findLocation method returns a Location

Parameters:
ref ReferenceType pointing to the class we have to put a breakpoint in

methodname Method name

line code line number

Returns:
a Location addressing a breakpoint. Location is a point within the executing code of the target VM.

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/monitor/BreakpointManager.java

B.23 it.cnr.isti.sel.server.ClassServer Class Reference

Inherited by it.cnr.isti.sel.server.ClassFileServer.

Public Member Functions

- abstract byte[ ] **getBytes** (String path) throws IOException, ClassNotFoundException
- void **run** ()

Protected Member Functions

- **ClassServer** (int port) throws IOException

B.23.1 Detailed Description

ClassServer(p. 158) is an abstract class that provides the basic functionality of a mini-webserver, specialized to load class files only. A **ClassServer**(p. 158) must be extended and the concrete subclass should define the **getBytes** method which is responsible for retrieving the bytecodes for a class.

The **ClassServer**(p. 158) creates a thread that listens on a socket and accepts HTTP GET requests. The HTTP response contains the bytecodes for the class that requested in the GET header.

For loading remote classes, an RMI application can use a concrete subclass of this server in place of an HTTP server.

See also:
ClassFileServer

B.23.2 Constructor & Destructor Documentation

B.23.2.1 it.cnr.isti.sel.server.ClassServer.ClassServer (int *port*) throws IOException [protected]

Constructs a **ClassServer**(p. 158) that listens on **port** and obtains a class's bytecodes using the method **getBytes**.

Parameters:
port the port number

Exceptions:
IOException if the **ClassServer**(p. 158) could not listen on **port**.

B.23.3 Member Function Documentation

B.23.3.1 abstract byte [ ] it.cnr.isti.sel.server.ClassServer.getBytes (String *path*) throws IOException, ClassNotFoundException [pure virtual]

Returns an array of bytes containing the bytecodes for the class represented by the argument **path**. The **path** is a dot separated class name with the ".class" extension removed.

Returns:
the bytecodes for the class

Exceptions:
ClassNotFoundException if the class corresponding to **path** could not be loaded.

IOException if error occurs reading the class

B.23.3.2 void it.cnr.isti.sel.server.ClassServer.run ()

The "listen" thread that accepts a connection to the server, parses the header to obtain the class file name and sends back the bytecodes for the class (or error if the class is not found or the response was malformed).

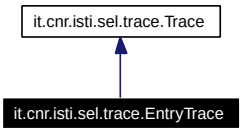
The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/server/ClassServer.java

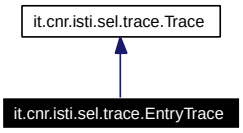
B.24 it.cnr.isti.sel.trace.EntryTrace Class Reference

Inherits **it.cnr.isti.sel.trace.Trace**.

Inheritance diagram for it.cnr.isti.sel.trace.EntryTrace:



Collaboration diagram for it.cnr.isti.sel.trace.EntryTrace:



Public Member Functions

- **EntryTrace** (MethodEntryEvent e, String outDir)
- void **manageEventInfo** ()

B.24.1 Detailed Description

Writes on a XML DOM file every information about the event that's been passed during construction

Author:
Annalisa Bacherotti

See also:
Trace(p. 182)

B.24.2 Constructor & Destructor Documentation

B.24.2.1 it.cnr.isti.sel.trace.EntryTrace.EntryTrace (MethodEntryEvent *e*, String *outDir*)

Class constructor

Parameters:
e event to monitor
outDir directory with output files

B.24.3 Member Function Documentation

B.24.3.1 void it.cnr.isti.sel.trace.EntryTrace.manageEventInfo () `[virtual]`

Manages this.entry_event writing necessary information on file nome_file and, if thread has been started from a remote connection, on file addressed by super.getRemoteCalls()

Implements **it.cnr.isti.sel.trace.Trace** (p. 183).

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/trace/EntryTrace.java

B.25 it.cnr.isti.sel.monitor.EventManager Class Reference

Public Member Functions

- **EventManager** (VirtualMachine vm, Vector< String > includes, Vector< String > interfacce, String outputDir, long ttime)
- void **run** ()
- String **createZip** () throws FileNotFoundException, IOException

Protected Member Functions

- void **setBreakpoint** (String className, String methodName, int line)

B.25.1 Detailed Description

Creates and requests events that will be monitored from target VM, put them in an EventQueue stack, then delegates EntryTrace and ExitTrace object instances to do the tracing of interesting methods. This class os the core of monitoring, because it initialize JDI functions to notify all interesting events, then responses with an appropriate function to events we are interesting on. These events are VmStartEvent, MethodEntry-Event, MethodExitEvent, ThreadtStartEvent, BreakpointEvent and VMDeathEvent. This event causes the end of monitoring

Author:
Annalisa Bacherotti Thread

B.25.2 Constructor & Destructor Documentation

B.25.2.1 it.cnr.isti.sel.monitor.EventManager.EventManager (VirtualMachine *vm*, Vector< String > *includes*, Vector< String > *interfacce*, String *outputDir*, long *ttime*)

Class constructor

Parameters:

vm Target Virtual Machine

includes string array containing the names of the packages

interfacce array di stringhe contenente i nomi delle interfacce da monitorare

outputDir directory contenente i file di output long value to name zip file with monitoring results

B.25.3 Member Function Documentation

B.25.3.1 String it.cnr.isti.sel.monitor.EventManager.createZip () throws FileNotFoundException, IOException

Creates a cmf<System.currentTimeMillis()>.zip file with xml files created in the tracing phase. For each file to be added, we have to: 1. create a zip entry 2. set entry size 3. set entry crc 4. add zipentry in the zipstream 5. write all bytes of the file in the zipstream

Returns:
name of created zip file

Exceptions:

FileNotFoundException

IOException

B.25.3.2 void it.cnr.isti.sel.monitor.EventManager.run ()

First initialize requests for interesting events, then implementes a loop to read this.events queue, extract an event and process it depending by its type and the importance of this event to the monitor, then exit this loop when VMDisconnectedException is thrown

See also:

initRequests

B.25.3.3 void it.cnr.isti.sel.monitor.EventManager.setBreakpoint (String *className*, String *methodName*, int *line*) `[protected]`

Puts a breakpoint at specified class, method and line in the target code

Parameters:

className class to put a breakpoint in

methodName method to put a breakpoint in

line line to put a breakpoint in

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/monitor/EventManager.java

B.26 it.cnr.isti.sel.server.EXaMSlave Class Reference

Public Member Functions

- **EXaMSlave** (int _port, int _httpport, String _classpath)

Static Public Member Functions

- static void **main** (String[ ] args)

Static Package Attributes

- static int **port** = 1099

  default port for RMI calls

- static int **http_port**

  port for HTTP server GET/SET calls

- static String **classpath**

  classpath where stub files are downloaded from

B.26.1 Detailed Description

This class starts up a minimal HTTP server where remote applications can download code from and found jars with stub classe for rmi calls. This jars are found through a codebase. Once running, your program can programmatically install a codebase property that points to the HTTP server running inside itself. One sets codebase to the HTTP directory, and the other can found code specified in the codebase here. So we need two port: one for RMI calls and the other for HTTP server. The class also starts up RMI server for RMI calls calling rebind, so that Setup stub and interface are read from HTTPServer and registerd in a rmiregistry. The need for a codebase and consequently for a HTTP server is that RMI protocol itself doesn't do the transmission of the code. Instead, RMI provides basic facilities to allow the receiver of an object to fetch the code for the object's class, if it's never seen it before and doesn't have it available locally.

Author:
Annalisa Bacherotti

B.26.2 Constructor & Destructor Documentation

B.26.2.1 it.cnr.isti.sel.server.EXaMSlave.EXaMSlave (int *_port*, int *_httpport*, String *_classpath*)

Class constructor

Parameters:

_port RMI server port

_httpport port where HTTP server is running on

_classpath directory where HTTP server is running on and where code ia available

B.26.3 Member Function Documentation

B.26.3.1 static void it.cnr.isti.sel.server.EXaMSlave.main (String[ ] *args*) [static]

EXaMSlave(p. 164) class starts up HTTP server on port and classpath supplied by input arguments, then binds rmi server to port 1099 for rmiregistry.

Parameters:

args first argument is port number for rmi calls; second argument is port number for HTTP server, from which remote applications can download code (codebase is set here); third argument is classpath where HTTP server is run (codebase is set here)

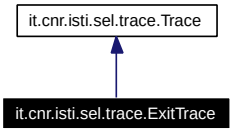
The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/server/EXaMSlave.java

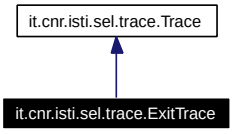
B.27 it.cnr.isti.sel.trace.ExitTrace Class Reference

Inherits **it.cnr.isti.sel.trace.Trace**.

Inheritance diagram for it.cnr.isti.sel.trace.ExitTrace:



Collaboration diagram for it.cnr.isti.sel.trace.ExitTrace:



Public Member Functions

- **ExitTrace** (MethodExitEvent e, String outDir)
- void **manageEventInfo** ()

B.27.1 Detailed Description

Writes on a XML DOM file every information about the event that's been passed during construction

Serializes event so that they can be written on a XML file. Once a DOM document has been obtained, we can write info about method we entered. A Method element contains:

- methName (string)
- param : - parName (string)- value (string)
- return (string)
- thisObject : - id (long)- class (string) and entering attribute. Then we insert parameter names and attribute class about each method, ignoring methods from Thread class that would throw. Eventually, return type and thisObject (if we aren't working on a statioc method) is written to XML file an AbsentInformationException. Once every information has been written, we can append method data to <thread> element in XML file, then serialize data and save file

Parameters:
nfile XML file

Exceptions:
Exception

Author:
Annalisa Bacherotti

See also:
Trace(p. 182)

B.27.2 Constructor & Destructor Documentation

B.27.2.1 it.cnr.isti.sel.trace.ExitTrace.ExitTrace (MethodExitEvent *e*, String *outDir*)

Class constructor

Parameters:

e event to monitor

outDir directory with output files

B.27.3 Member Function Documentation

B.27.3.1 void it.cnr.isti.sel.trace.ExitTrace.manageEventInfo () [virtual]

Manages exit_event writing necessary information on file nome_file and, if thread has been started from a remote connection, on file addressed by super.getRemoteCalls()

Implements **it.cnr.isti.sel.trace.Trace** (p. 183).

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/trace/ExitTrace.java

B.28 it.cnr.isti.sel.exceptions.JarConnectionException Class Reference

Public Member Functions

- **JarConnectionException** ()

  Class constructor, that only calls super class constructor passing msg string to it.

B.28.1 Detailed Description

This class manages exception form reading MANIFEST file in jar files, when trying to retrieve main fucntion name

Author:
Annalisa Bacherotti

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/exceptions/JarConnectionException.java

B.29 it.cnr.isti.sel.utils.JarDownloadManager Class Reference

Static Public Member Functions

- static void **downloadJar** (String machine, int port, String folderStorage, String jarName) throws JarConnectionException

Static Protected Attributes

- static final String **fileSeparator** = System.getProperty("file.separator")

  file separator independent from operating system

B.29.1 Detailed Description

This class allows downloading of files from listener server on EXaMMaster.

Author:
Annalisa Bacherotti

B.29.2 Member Function Documentation

B.29.2.1 static void it.cnr.isti.sel.utils.JarDownloadManager.downloadJar (String *machine*, int *port*, String *folderStorage*, String *jarName*) throws JarConnectionException [static]

Downloads jarName from specified host:port/folderStorage

Parameters:

machine host to download files from

port port on machine to connect to

folderStarage folder where downloaded files are stored

jarName file to download

Exceptions:

JarConnectionException

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/utils/JarDownloadManager.java

B.30 it.cnr.isti.sel.utils.JarProperties Class Reference

Public Member Functions

- **JarProperties** (String jarfile) throws IOException
- String **getMainClass** () throws IOException

B.30.1 Detailed Description

This class manages jar file properties

Author:
Annalisa Bacherotti

B.30.2 Constructor & Destructor Documentation

B.30.2.1 it.cnr.isti.sel.utils.JarProperties.JarProperties (String *jarfile*) throws IOException

Class constructor

Parameters:
jarfile jarfile to construct object to

Exceptions:
IOException

B.30.3 Member Function Documentation

B.30.3.1 String it.cnr.isti.sel.utils.JarProperties.getMainClass () throws IOException

Returns:
main function

Exceptions:
IOException

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/utils/JarProperties.java

B.31 it.cnr.isti.sel.utils.MethodListObj Class Reference

Public Member Functions

- **MethodListObj** (String n, List< Method > metodi)
- String **getNome** ()
- boolean **isMethodIn** (String firmaMetodo)

B.31.1 Detailed Description

Object within this class are used to store methods from interfaces that have to be monitored (those methods are store in this.meth_list) then matched to interfaces where they have been declared. Once an object has been constructed, it can't be modified.

Author:
Alessia Bardi

B.31.2 Constructor & Destructor Documentation

B.31.2.1 it.cnr.isti.sel.utils.MethodListObj.MethodListObj (String *n*, List< Method > *metodi*)

Class constuctor

Parameters:
n interface name

metodi com.sun.jdi.Method objects list

B.31.3 Member Function Documentation

B.31.3.1 String it.cnr.isti.sel.utils.MethodListObj.getNome ()

Returns:
this.nomeInterface

B.31.3.2 boolean it.cnr.isti.sel.utils.MethodListObj.isMethodIn (String *firmaMetodo*)

Searches in this.methList for a method whose signature is firmaMetodo

Parameters:
firmaMetodo method signature

Returns:
true if method with firmaMetodo signature is in list, false otherwise

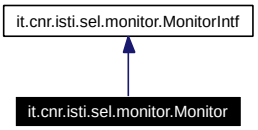
The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/utils/MethodListObj.java

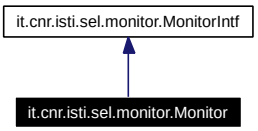
B.32 it.cnr.isti.sel.monitor.Monitor Class Reference

Inherits **it.cnr.isti.sel.monitor.MonitorIntf**.

Inheritance diagram for it.cnr.isti.sel.monitor.Monitor:



Collaboration diagram for it.cnr.isti.sel.monitor.Monitor:



Public Member Functions

- **Monitor** (String main, String args, String options, Vector< String > include, Vector< String > interfaces, String outputDir, long ttime)
- void **setBreakPoint** (String className, String methodName, int lineCode)
- void **launchMonitor** ()
- void **upload** (String machine, int port) throws IOException
- void **upload** (String fileToUpload, String machine, int port) throws IOException

B.32.1 Detailed Description

Starts monitoring of target application using a JDI LaunchingConnector. Target application launched is addressed by main function and args parameter. Target application is launched using java command and appropriate options passed by Setup class. When target parameter has been launched, registers output, JVM information and errors if any in specific vectors. This class Implements MonitrorIntf interface

Author:
Annalisa Bacherotti

See also:
MonitorIntf(p. 175)

B.32.2 Constructor & Destructor Documentation

B.32.2.1 it.cnr.isti.sel.monitor.Monitor.Monitor (String *main*, String *args*, String *options*, Vector< String > *include*, Vector< String > *interfaces*, String *outputDir*, long *ttime*)

Class constructor

Parameters:
main function

args main arguments

options VM options

include Packages that will be monitored

interfaces Interfaces that will be monitored

outputDir Directory where results will be stored

B.32.3 Member Function Documentation

B.32.3.1 void it.cnr.isti.sel.monitor.Monitor.launchMonitor ()

Starts monitoring of target application using a JDI LaunchingConnector The target application launched is addressed by main function and args parameter

Implements **it.cnr.isti.sel.monitor.MonitorIntf** (p. 175).

B.32.3.2 void it.cnr.isti.sel.monitor.Monitor.setBreakPoint (String *className*, String *methodName*, int *lineCode*)

Sets a breakpoint in classname class, at methodname method and lineCode line

Parameters:

classname class name

methodName method name

lineCode code line number

Implements **it.cnr.isti.sel.monitor.MonitorIntf** (p. 175).

B.32.3.3 void it.cnr.isti.sel.monitor.Monitor.upload (String *fileToUpload*, String *machine*, int *port*) throws IOException

This function allows to upload fileToUpload file on EXaMMaster machine:port

Parameters:

fileToUpload file to be uploaded

machine Host to upload requested jar to

port Port where listener is active

Exceptions:

IOException

Implements **it.cnr.isti.sel.monitor.MonitorIntf** (p. 176).

B.32.3.4 void it.cnr.isti.sel.monitor.Monitor.upload (String *machine*, int *port*) throws IOException

This function allows to upload the requested jar on EXaMMaster machine:port

Parameters:

machine Host to upload requested jar to

port Port where listener is active

Exceptions:
IOException

Implements **it.cnr.isti.sel.monitor.MonitorIntf** (p. 176).

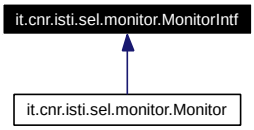
The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/monitor/Monitor.java

B.33 it.cnr.isti.sel.monitor.MonitorIntf Interface Reference

Inherited by **it.cnr.isti.sel.monitor.Monitor**.

Inheritance diagram for it.cnr.isti.sel.monitor.MonitorIntf:



Public Member Functions

- void **launchMonitor** ()
- void **setBreakPoint** (String className, String methodName, int lineCode)
- void **upload** (String machine, int port) throws IOException
- void **upload** (String fileToUpload, String machine, int port) throws IOException

B.33.1 Detailed Description

Interface to export monitoring function. If we want to change monitoring, we just have to change **Monitor**(p. 172) class with another that implements functions listed below

Author:
Annalisa Bacherotti

B.33.2 Member Function Documentation

B.33.2.1 void it.cnr.isti.sel.monitor.MonitorIntf.launchMonitor ()

Starts monitoring of target application using a JDI LaunchingConnector The target application launched is addressed by main function and args parameter

Implemented in **it.cnr.isti.sel.monitor.Monitor** (p. 173).

B.33.2.2 void it.cnr.isti.sel.monitor.MonitorIntf.setBreakPoint (String *className*, String *methodName*, int *lineCode*)

Sets a breakpoint in classname class, at methodname method and lineCode line

Parameters:
classname class name

methodName method name

lineCode code line number

Implemented in **it.cnr.isti.sel.monitor.Monitor** (p. 173).

B.33.2.3 void it.cnr.isti.sel.monitor.MonitorIntf.upload (String *fileToUpload*, String *machine*, int *port*) throws IOException

This function allows to upload fileToUpload file on EXaMMaster machine:port

Parameters:

fileToUpload file to be uploaded

machine Host to upload requested jar to

port Port where listener is active

Exceptions:

IOException

Implemented in **it.cnr.isti.sel.monitor.Monitor** (p. 173).

B.33.2.4 void it.cnr.isti.sel.monitor.MonitorIntf.upload (String *machine*, int *port*) throws IOException

This function allows to upload the requested jar on EXaMMaster machine:port

Parameters:

machine Host to upload requested jar to

port Port where listener is active

Exceptions:

IOException

Implemented in **it.cnr.isti.sel.monitor.Monitor** (p. 173).

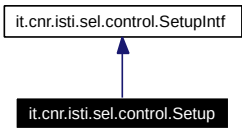
The documentation for this interface was generated from the following file:

- src/it/cnr/isti/sel/monitor/MonitorIntf.java

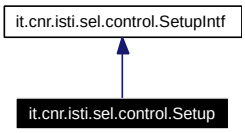
B.34 it.cnr.isti.sel.control.Setup Class Reference

Inherits **it.cnr.isti.sel.control.SetupIntf**.

Inheritance diagram for it.cnr.isti.sel.control.Setup:



Collaboration diagram for it.cnr.isti.sel.control.Setup:



Public Member Functions

- **Setup** () throws RemoteException
- void **setBreakPoint** (String className, String methodName, int lineCode) throws RemoteException
- void **launchTarget** (String examIP, int examJarPort, String jarname, Vector< String > arguments, String options, Vector< String > include, Vector< String > interfaces, String tmpDir, long ttime) throws RemoteException

B.34.1 Detailed Description

OVERVIEW: This class, whose methods can be invoked remotely using its public interface **SetupIntf**(p. 179), allows EXaMSlave to locally launch a Virtual Machine on the target machine. It's invoked by EXaMMaster to give directions about monitoring. When monitoring is over, uploads zip files containing results and dat file with standard output to EXaMMaster host.

Author:
Annalisa Bacherotti

See also:
SetupIntf(p. 179)

B.34.2 Constructor & Destructor Documentation

B.34.2.1 it.cnr.isti.sel.control.Setup.Setup () throws RemoteException

Default Class constructor. Just call for super class constructor

Exceptions:
RemoteException

B.34.3 Member Function Documentation

B.34.3.1 void it.cnr.isti.sel.control.Setup.launchTarget (String *examIP*, int *examJarPort*, String *jarname*, Vector< String > *arguments*, String *options*, Vector< String > *include*, Vector< String > *interfaces*, String *tmpDir*, long *ttime*) throws RemoteException

This method downloads target application compressed in a jar file from examIP host at examJarPort, then launches target application and, if requested, launches monitor obtaining a reference to the target Virtual Machine. Then manages displaying of messages resulting from monitoring, redirecting standard output to a file addressed by ttime long value. Class name containing main function is read from MANIFEST file in jarname .jar file. When monitoring is over, it uploads resulting zip file and output dat file to exam master.

Parameters:

IP address of master host, necessary to download jar file from

examJarPort port to downloaded jar file from (for example `http://146.48.84.135:8001/jars`)

jarname the jar containing target application

arguments name of the class containing main function and its parameters

options virtual machine options (-cp classpath)

include string array containing the names of the packets to monitor

interfaces string array containing the names of the interfaces whose methods has to be monitored (they must belong to one of the packets mentioned in include array)

tmpDir directory where monitoring output files are stored long value to give name to zip file containing results, so that it can be uniquely identified

Exceptions:

RemoteException

Implements **it.cnr.isti.sel.control.SetupIntf** (p. 179).

B.34.3.2 void it.cnr.isti.sel.control.Setup.setBreakPoint (String *className*, String *methodName*, int *lineCode*) throws RemoteException

Allows to put a breakpoint at specified class, method, line number

Parameters:

class to put a breakpoint in

method to put a breakpoint in

code line number to put a breakpoint in

Exceptions:

RemoteException

Implements **it.cnr.isti.sel.control.SetupIntf** (p. 180).

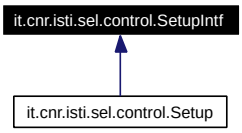
The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/control/Setup.java

B.35 it.cnr.isti.sel.control.SetupIntf Interface Reference

Inherited by **it.cnr.isti.sel.control.Setup**.

Inheritance diagram for it.cnr.isti.sel.control.SetupIntf:



Public Member Functions

- void **launchTarget** (String teseoIP, int teseoJarPort, String jarname, Vector< String > arguments, String options, Vector< String > include, Vector< String > interfaces, String tmpDir, long ttime) throws RemoteException
- void **setBreakPoint** (String className, String methodName, int lineCode) throws RemoteException

B.35.1 Detailed Description

Interface with public methods to remotely invoke methods from **Setup**(p. 177) class, a class implementing this interface.

Author:
Annalisa Bacherotti

See also:
Setup(p. 177)

B.35.2 Member Function Documentation

B.35.2.1 void it.cnr.isti.sel.control.SetupIntf.launchTarget (String *teseoIP*, int *teseoJarPort*, String *jarname*, Vector< String > *arguments*, String *options*, Vector< String > *include*, Vector< String > *interfaces*, String *tmpDir*, long *ttime*) throws RemoteException

This method downloads target application compressed in a jar file from examIP host at examJarPort, then launches target application and, if requested, launches monitor obtaining a reference to the target Virtual Machine. Then manages displaying of messages resulting from monitoring, redirecting standard output to a file addressed by ttime long value. Class name containing main function is read from MANIFEST file in jarname .jar file. When monitoring is over, it uploads resulting zip file and output dat file to exam master.

Parameters:

IP address of master host, necessary to download jar file from

examJarPort port to downloaded jar file from (for example `http://146.48.84.135:8001/jars`)

jarname the jar containing target application

arguments name of the class containing main function and its parameters

options virtual machine options (-cp classpath)

include string array containing the names of the packets to monitor

interfaces string array containing the names of the interfaces whose methods has to be monitored (they must belong to one of the packets mentioned in include array)

tmpDir directory where monitoring output files are stored long value to give name to zip file containing results, so that it can be uniquely identified

Exceptions:

RemoteException

Implemented in **it.cnr.isti.sel.control.Setup** (p. 178).

B.35.2.2 void it.cnr.isti.sel.control.SetupIntf.setBreakPoint (String *className*, String *methodName*, int *lineCode*) throws RemoteException

Places a breakpoint at the given class, method and line number of target application

Parameters:

className class breakpoint has to be put in

methodName method, within className, breakpoint has to be put in

lineCode line, within methodName, breakpoint has to be put in

Exceptions:

RemoteException

Implemented in **it.cnr.isti.sel.control.Setup** (p. 178).

The documentation for this interface was generated from the following file:

- src/it/cnr/isti/sel/control/SetupIntf.java

B.36 it.cnr.isti.sel.outputlistener.TargetListener Class Reference

Public Member Functions

- **TargetListener** (InputStream stream, String nome)
- void **run** ()
- Vector< String > **getOutput** ()

B.36.1 Detailed Description

This class manages redirection of target application output preventing its stream buffers to overflow and application to halt during execution.

Author:
Alessia Bardi

B.36.2 Constructor & Destructor Documentation

B.36.2.1 it.cnr.isti.sel.outputlistener.TargetListener.TargetListener (InputStream *stream*, String *nome*)

Class constructor

Parameters:
stream stream where application can read data

nome class name

B.36.3 Member Function Documentation

B.36.3.1 Vector<String> it.cnr.isti.sel.outputlistener.TargetListener.getOutput ()

Returns:
output vector

B.36.3.2 void it.cnr.isti.sel.outputlistener.TargetListener.run ()

The stream is read out and each stream of data is printed to screen

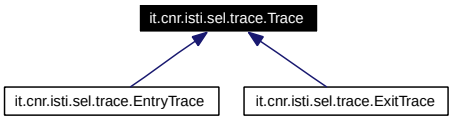
The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/outputlistener/TargetListener.java

B.37 it.cnr.isti.sel.trace.Trace Class Reference

Inherited by **it.cnr.isti.sel.trace.EntryTrace**, and **it.cnr.isti.sel.trace.ExitTrace**.

Inheritance diagram for it.cnr.isti.sel.trace.Trace:



Public Member Functions

- **Trace** (String outDir, ThreadReference **tr**)

  Unique ID of thread managing end of remote calls.

- String **getRemoteCalls** ()
- long **getRemoteThreadId** ()
- long **getDestroyVMThreadId** ()
- abstract void **manageEventInfo** ()
- void **serializeOutput** (Document doc, String nfile)
- ObjectReference **getInstanceMethodObject** ()

Static Public Member Functions

- static void **printStack** (ThreadReference t, String outputDir) throws IOException

Protected Member Functions

- void **traceCallToRemoteHost** (ThreadReference **tr**) throws IOException
- void **traceRemoteCalls** (ThreadReference **tr**) throws IOException

Protected Attributes

- ThreadReference **tr**

  Thread reference.

- String **nome_file**

  File name for thread traces.

B.37.1 Detailed Description

Abstract class that provides methods to manage traceable events. it also treats threads generated by TCP Connection related to remote calls

Author:
Annalisa Bacherotti

See also:
EntryTrace(p. 160) **ExitTrace**(p. 166)

B.37.2 Constructor & Destructor Documentation

B.37.2.1 it.cnr.isti.sel.trace.Trace.Trace (String *outDir*, ThreadReference *tr*)

Unique ID of thread managing end of remote calls.

Class constructor

Parameters:
outDir output directory

tr reference to the thread whose execution we have to trace

B.37.3 Member Function Documentation

B.37.3.1 long it.cnr.isti.sel.trace.Trace.getDestroyVMThreadId ()

Returns:
Id of file that stores information about end of remote calls

B.37.3.2 ObjectReference it.cnr.isti.sel.trace.Trace.getInstanceMethodObject ()

Returns:
o ObjectReference to the object that current Stackframe belongs to or null if it doesn-t exist for current StackFrame or if this.tr is not in a compatible state (if it is a zombie or if it is in a sleep state)

B.37.3.3 String it.cnr.isti.sel.trace.Trace.getRemoteCalls ()

Returns:
Name of file that stores information about remote calls

B.37.3.4 long it.cnr.isti.sel.trace.Trace.getRemoteThreadId ()

Returns:
ID of file that stores information about remote calls

B.37.3.5 abstract void it.cnr.isti.sel.trace.Trace.manageEventInfo () `[pure virtual]`

Gathers and manages information about e event

Implemented in **it.cnr.isti.sel.trace.EntryTrace** (p. 160), and **it.cnr.isti.sel.trace.ExitTrace** (p. 167).

B.37.3.6 static void it.cnr.isti.sel.trace.Trace.printStack (ThreadReference *t*, String *outputDir*) throws IOException [static]

Standard outputs stack of thread t and values of variables

Parameters:
t Reference to the thread whose stack has to be printed to screen

Exceptions:
IOException

B.37.3.7 void it.cnr.isti.sel.trace.Trace.serializeOutput (Document *doc*, String *nfile*)

serializes XML document so that it can be written to a file

Parameters:
doc document to serialize

B.37.3.8 void it.cnr.isti.sel.trace.Trace.traceCallToRemoteHost (ThreadReference *tr*) throws IOException [protected]

If thread just started contains "DestroyJavaVM" string in his name, then this thread ends a remote connection. If so, a new file is created, if it doesn't exists yet, with a name that points to explain who is calling who: caller_calls_callee_currentTimeMillis.xml.

Parameters:
tr Thread Reference to check

Exceptions:
IOException

B.37.3.9 void it.cnr.isti.sel.trace.Trace.traceRemoteCalls (ThreadReference *tr*) throws IOException [protected]

If thread just started contains "RMI TCP Connection" string in his name, then this thread has been started by a remote connection. If so, a new file is created, if it doesn't exists yet, with a name that points to who is calling who: caller_calls_callee_currentTimeMillis.xml.

Parameters:
tr Thread Reference to check

Exceptions:
IOException

The documentation for this class was generated from the following file:

- src/it/cnr/isti/sel/trace/Trace.java