UNIVERSITÀ DEGLI STUDI DELLA BASILICATA

DIPARTIMENTO DI INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

E DELLE TECNOLOGIE DELL'INFORMAZIONE

RELAZIONE PROGETTO FINALE TAV

ASFJ: A Simple Firewall in Java

Docente:

Ch.mo Prof. Gianvito Summa

Studente:

Michael Pio Stolfi 68787

ANNO ACCADEMICO 2024-2025

Sommario

Introduzione

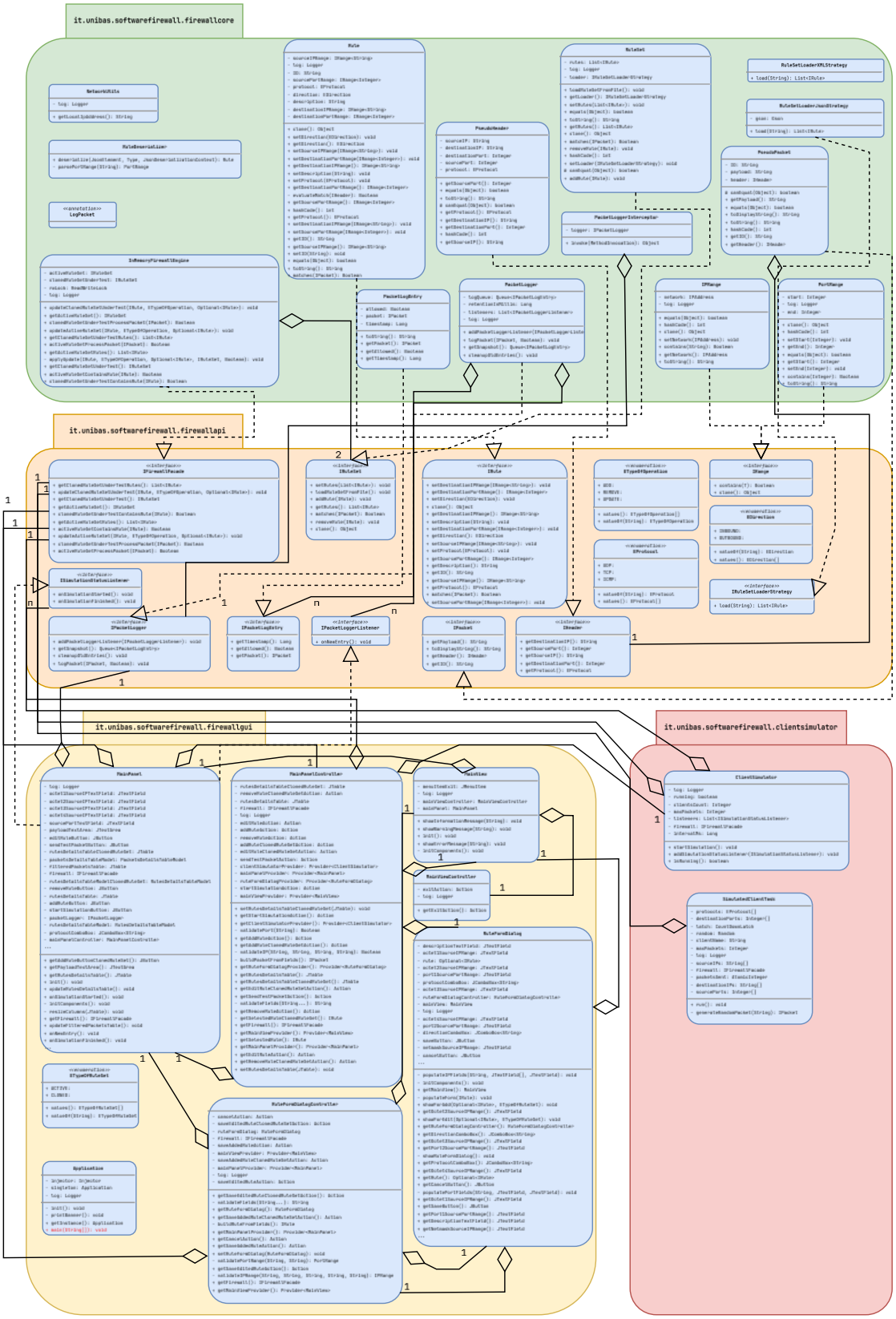
A Simple Firewall in Java (ASFJ) è un'applicazione Java desktop modulare che simula il comportamento di un firewall software. Il progetto permette di testare, monitorare e modificare dinamicamente regole di filtraggio su pacchetti generati da client concorrenti. L'interfaccia utente, costruita con Swing, consente un'interazione in tempo reale con il traffico e le regole, supportando anche la simulazione di ambienti di rete controllati per fini didattici o sperimentali.

ASFJ nonostante abbia fini didattici è stato sviluppato con un'attenzione particolare a: tenere alto l'isolamento aumentando la coesione (tramite la separazione dei moduli), all'estendibilità riducendo l'accoppiamento (tramite l'utilizzo estensivo delle interfacce), e all'applicazione dei principi avanzati dell'ingegneria del software, come l'Inversion of Control (IoC), i Design Pattern (DP), e la personalizzazione del comportamento applicativo tramite file properties. Per tale applicativo come in tutti gli ambienti professionali non si è ignorato anche la importante fase finale dello sviluppo applicativo ovvero la fase di deploy. A tale scopo esso è stato impacchettato sotto forma di uber-JAR; un uber-JAR, noto anche come fat JAR, è un file JAR che contiene non solo un programma Java, ma anche le sue dipendenze. Ciò significa che il JAR funziona come una distribuzione "all-in-one" del software, senza bisogno di altro codice Java.

Ovviamente questo approccio estremamente professionale allo sviluppo ha un chiaro inconveniente ovvero che l'architettura applicativa si complica e diviene per un osservatore esterno di difficile comprensione. Questo documento ha come fine: l'integrazione del README di progetto, la spiegazione dell'architettura applicativa, l'esposizione dei dettagli applicativi complessi e infine la discussione di come sono stati implementati gli argomenti centrali del progetto ovvero: Dependency Injection (DI), Aspect Oriented Programming (AOP), Thread e Sincronizzazione, e Clonazione.

Architettura applicativa

Per la spiegazione dell'architettura applicativa è stato realizzato l'UML Class Diagram di tutto il progetto, consultabile nell'immagine sottostante:



In questo Class Diagram sono state volutamente tralasciate le relazioni di tipo "dipendenza" perché avrebbero reso esternamente complesso il grafico, il quale sarebbe risultato in una inutile foresta di collegamenti di difficile consultazione. Si sono preservati invece le relazioni di tipo "implementazione" poiché chiariscono i ruoli dei componenti e le relazioni forti come aggregazione e composizione. Come è possibile notare quest'ultima relazione ovvero la composizione nonostante non sia stata ignorata non figura nemmeno una volta nel grafico. Questo è dovuto al fatto che la relazione di composizione (canonicamente rappresentata in UML con una freccia con una estremità a diamante pieno) è di fatto come una relazione di aggregazione con la differenza che la classe che mantiene il riferimento all'altra classe ne gestisce anche il ciclo di vita. Questo evento non avviene in ASFJ perché per lo sviluppo dell'applicativo si è adottato un approccio DI-First ovvero la Dependency Injection è stata applicata ovunque possibile per ridurre l'accoppiamento e di conseguenza tutte le classi sono gestire dall'Injector e in genere il loro ciclo di vita non è gestito da altre classi.

ASFJ è strutturata in quattro moduli (corrispondenti ad altrettanti sotto-progetti Gradle): **firewall-api**, **firewall-core**, **firewall-gui** e **client-simulator**.

Il modulo **firewall-api** contiene le interfacce condivise tra i moduli e rappresenta il contratto che lega insieme gli altri componenti e consente un disaccoppiamento chiaro tra implementazioni e dipendenze. Come è possibile notare dal Class Diagram grazie a firewall-api che funge da "layer di comunicazione" tra i moduli non vi è alcun collegamento ad esempio tra i moduli firewall-gui e firewall-core. Questo ha il chiaro vantaggio che rende ognuno dei moduli implementativi facilmente sostituibili nella loro totalità o anche solo in alcuni dei loro componenti. Se ad esempio si è interessati a una interfaccia di tipo diverso come una CLI o a una FE Web è possibile sostituire il modulo firewall-gui che per ora corrisponde a una interfaccia desktop in Swing con il modulo che più si desidera e l'applicativo continuerà a funzionare. Come è possibile notare la maggioranza dei collegamenti che vengono dal modulo firewall-

core verso firewall-api sono di tipo "implementazione" siccome nel core risiedono molte delle implementazioni di questo modulo. Mentre la maggioranza dei collegamenti che vengono dal modulo firewall-gui verso firewall-api sono di tipo "aggregazione" e questo risulta naturale se si pensa che di fatto è poi la GUI l'attore che in risposta alle azioni dell'utente lancia le computazioni applicative. È inoltre interessante notare come in risposta a una crescente complessità della stessa API si è introdotto il pattern Façade incarnato dall'interfaccia IFarewallFacade, la quale rappresenta un punto unico di accesso alla maggioranza delle funzionalità applicative per i moduli esterni. La creazione di IFarewallFacade ha il chiaro vantaggio di semplificare il modo in cui i moduli esterni si interfacciano con l'API, fermo restando che nel caso della necessità di utilizzo di una funzionalità più avanzata non coperta dal Façade si può e si deve comunque interfacciarsi con una interfaccia normale.

Il modulo **firewall-core** rappresenta l'attuale implementazione In-Memory del firewall, e implementa la logica centrale del firewall, compresi: la gestione dei set di regole (caricamento da file, test su varianti del set), la gestione delle regole di filtraggio (aggiunta, modifica e rimozione), l'ispezione dei pacchetti filtrati. In questo modulo vi sono: Utility Class come RuleDeserializer e NetworkUtils; Domain Model Class quindi classi che combinano dati e metodi come RuleSet e Rule; e infine ci sono due importanti Service Class ovvero InMemoryFirewallEngine e PacketLogger che sono Singleton e utilizzando le altre classi incapsulano la logica di business, permettendo il funzionamento dell'applicativo. InMemoryFirewallEngine implementa l'interfaccia centrale IFirewallFacade e ne rappresenta una implementazione totalmente Thread-Safe poiché ogni metodo è sincronizzato con un lock che distingue scritture da letture in modo da ottenere anche le massime performance. Quindi se un modulo esterno utilizza IFirewallFacade e di conseguenza InMemoryFirewallEngine per effettuare le sue operazioni avrà la certezza di effettuare operazioni stabili a modifiche concorrenti senza doversi gestire la complessità collegata. PacketLogger implementa l'interfaccia IPacketLogger e rappresenta un punto unico e centralizzato per il logging

dei pacchetti filtrati dall'applicativo. Si può richiedere il logging al PacketLogger sia tramite un Interceptor sia tramite una tradizionale chiamata di metodo. Esso una volta loggato un pacchetto si occupa anche di notificare l'arrivo di una nuova entry a una serie di sottoscrittori implementando difatti un Observer Pattern. È infine interessante notare come si sia adottato uno Strategy Pattern per il caricamento delle regole dal disco poiché in questo modulo ci sono due classi che implementano IRuleSetLoaderStrategy ovvero RuleSetLoaderJsonStrategy e RuleSetLoaderXMLStrategy che permettono di caricare i RuleSet sia da file JSON che da file XML. La scelta di quale delle due implementazioni si debba utilizzare è fatta per permettere la massima configurabilità tramite delle properties. Normalmente per la scelta di quale implementazione utilizzare ovvero nel caso di implementazione condizionale, come avviene per ASFJ, viene utilizzata una Factory apposita. In tal caso però tale Factory non esiste dato che della creazione degli oggetti se ne occupa l'Injector, e per permettere la logica di binding condizionale si è implementato un particolare metodo annotato con @Provides.

Il modulo **firewall-gui** presenta classi dedite alla visualizzazione dei componenti a schermo (View) come MainPanel e RuleFormDialog, classi dedite al controllo ovvero alle azioni da effettuare quando un evento delle viste viene scatenato (Controller) come MainPanelController, e infine classi utili ad esporre dati per la GUI (Presentation Model) come RulesDetailsTableModel. Quest'ultime sono piuttosto standard e di conseguenza poco interessanti da discutere. Una nota a parte merita la classe Application poiché nonostante non si occupi più della Service Location applicativa (quel pattern è stato abbandonato in favore della DI pura) effettua una serie di compiti fondamentali per il funzionamento dell'applicazione. Difatti Application: contiene il main e di conseguenza rappresenta il punto di esecuzione dell'applicazione; avvia e configura la MainView ovvero il JFrame su cui si poggiano tutti i pannelli; infine si occupa anche di creare l'Injector di Guice e di conseguenza è anche l'unica classe che dipende direttamente dall'Injector.

Infine il modulo **client-simulator** si occupa di simulare diversi client che concorretemene contattano il firewall. La simulazione avviene nella classe ClientSimulator tramite diversi thread creati tramite un thread-pool. Numero di thread, numero di pacchetti da inviare per ogni thread e intervallo di attesa tra l'invio di un pacchetto e il successivo sono tutti parametri configurabili da poperties. La creazione dei pacchetti pseudo-casuali e l'invio dei pacchetti al firewall avviene però nella classe SimulatedClientTask che implementa Runnable e difatti rappresenta l'azione eseguita da ogni thread. Questo modulo è contattato dal MainPanelController della GUI che infatti è l'unica classe che ne detiene un riferimento.

Dependency Injection

Come Framework di Dependency Injection si è utilizzato Google Guice, nella sua ultima versione ovvero la v7.0.0. Come accennato in precedenza la creazione dell'Injector di Guice viene effettuata nella classe Application con la seguente riga di codice:

```
private final Injector injector;

private Application() {
  …
  this.injector = Guice.createInjector(stage, new FirewallGUIModule(),
                                               new FirewallCoreModule(),
                                               new ClientSimulatorModule());
}
```

In questo modo Guice prede in input uno o più moduli (Module), che specificano le regole di binding (cioè, cosa iniettare dove). Siccome Guice di default non fa autowiring è necessario specificargli quale classe concreta è necessario iniettare quando viene richiesta una determinata interfaccia. Le classi FirewallGUIModule, FirewallCoreModule, ClientSimulatorModule, sono classi che estendendo AbstractModule e dentro contengono metodi `configure()` dove si definisce quale implementazione usare per ogni interfaccia, o come costruire determinati oggetti. Si è provveduto a creare un modulo Guice per ogni modulo applicativo. Nei metodi `configure()` è possibile trovare configurazioni come le seguenti:

```
@Override
    protected void configure() {
        this.bind(IFirewallFacade.class).to(InMemoryFirewallEngine.class);
        this.bind(IRuleSet.class).to(RuleSet.class);
        this.bind(IPacketLogger.class).to(PacketLogger.class);
        …
   }
```

Che come detto si occupano di mappare le interfacce con le corrette implementazioni ma vengano effettuate anche configurazioni meno comuni come le seguenti:

```
@Override
    protected void configure() {
        bind(RulesDetailsTableModel.class)
            .annotatedWith(Names.named("active"))
            .to(RulesDetailsTableModel.class)
            .in(Singleton.class);

        bind(RulesDetailsTableModel.class)
            .annotatedWith(Names.named("cloned"))
            .to(RulesDetailsTableModel.class)
            .in(Singleton.class);

    }
```

Che permette di ottenere due Singleton distinti (annotati con un differente tag) della stessa classe concreta ovvero RulesDetailsTableModel, che inoltre essendo una Presentation Model Class non ha nemmeno una interfaccia dedicata. Ovunque era possibile si è preferito utilizzare le annotazioni per assegnare gli scope alle classi (tipo `@Singleton` posta prima della dichiarazione della classe) poiché rispetto ad inserire lo scope all'interno del modulo di configurazione di Guice hanno il chiaro vantaggio di essere sempre presenti insieme alla classe e quindi di chiarire subito il suo scope. In queste classi vi sono però non solo metodi *`configure()`* ma anche metodi speciali annotati con `@Provides` che servono a definire manualmente come costruire un'istanza di un certo tipo, invece di usare `bind()` in `configure()` questo perché il metodo classico non permette logica condizionale:

```
    @Provides
    public IRuleSetLoaderStrategy provideRuleSetLoaderStrategy() {
        Properties props = new Properties();
        try (InputStream in = this.getClass()

                        .getClassLoader()

                        .getResourceAsStream("firewall-core.properties")) {
            props.load(in);
            String loaderType = props

                            .getProperty("firewallcore.ruleset.loader", "JSON");
            if ("XML".equalsIgnoreCase(loaderType)) {
                return new RuleSetLoaderXMLStrategy();
            }
        } catch (IOException e) {
            log.error("Could not load rule set loader configuration: ", e);
        }
        return new RuleSetLoaderJsonStrategy(); // default fallback
    }
```

Finita la fase di configurazione di Guice è possibile finalmente utilizzarlo. Guice supporta tutti i tipi di iniezione, ma raccomanda fortemente quella tramite costruttore, perché è la più sicura e mantenibile nel lungo periodo, specialmente in progetti di grandi dimensioni. Questo perché: si vede subito da cosa dipende la classe; non è possibile dimenticare una dipendenza; supporta l'uso di campi final; l'oggetto non cambia dopo la costruzione; e c'è un migliore supporto da parte di tool/test/mock framework. Di conseguenza ovunque era possibile si è preferita questa strategia di iniezione nello sviluppo dell'applicativo. Un esempio di utilizzo di Guice per l'iniezione di una dipendenza tramite costruttore è riportato di seguito:

```
@Inject
public InMemoryFirewallEngine(IRuleSet ruleSet) {
    this.activeRuleSet = ruleSet;
}
```

È interessante notare come anche in questo caso si avvisa Guice della necessaria iniezione della dipendenza tramite un'apposita annotazione.

È inoltre interessante discutere come è stato risolto il problema delle dipendenze cicliche. Ci sono infatti dei casi comuni in cui due classi dipendono vicendevolmente. Ad esempio nel contesto di ASFJ il MainPanel dipende dal MainPanelController siccome deve settare le azioni ai suoi pulsanti associandole alle corrispettive azioni definite nel MainPanelController, a sua volta però il MainPanelController dipende dal MainPanel siccome tra le altre cose deve ad esempio aggiornare le tabelle una volta effettuate le operazioni. Questo è un caso tipico di dipendenza ciclica (A -> B -> A), se si fosse effettuata l'iniezione delle dipendenze nella maniera classica Guice avrebbe rilanciato eccezione poiché non avrebbe saputo risolvere il grafo delle dipendenze. Ci sono principalmente tre soluzioni a questo problema: una prima soluzione è quella di cambiare strategia di iniezione per la proprietà in questione passando ad esempio a una iniezione tramite proprietà o tramite metodo setter che quindi avviene solo dopo la creazione dell'oggetto della classe; una seconda soluzione consiste nel chiedere esplicitamente a Guice tramite il metodo `getInstance(…)` di Injector la dipendenza desiderata solo

quando serve quindi non chiedendola alla creazione dell'oggetto della classe ma nel metodo di interesse; infine una terza soluzione è quella di utilizzare una interfaccia fornita da Guice e preposta a risolvere questi problemi ovvero `Provider<T>` che serve per demandare la creazione di un oggetto a un momento successivo, invece di farlo subito all'iniezione. Si è scelta la terza soluzione poiché la prima soluzione presenta l'inconveniente di costringere a mischiare due stili diversi di iniezione e inoltre a usare uno stile non consigliato; mentre la seconda presenta l'inconveniente di legare un'altra classe all'Injector di Guice e inoltre se una classe chiama `injector.getInstance(...)`, sta lei stessa decidendo da dove prendere le dipendenze, questo è un Service Locator Pattern, si sta rompendo il principio di base della Dependency Injection. Il Provider è difatti un segnaposto che permette poi di accedere all'oggetto desiderato nel momento richiesto tramite il metodo `get()` come fatto nel seguente snippet di codice:

```
private final IFirewallFacade firewall;
private final Provider<MainPanel> mainPanelProvider;
@Inject
public MainPanelController(IFirewallFacade firewall,
                           Provider<MainPanel> mainPanelProvider,…){
    this.firewall = firewall;
    this.mainPanelProvider = mainPanelProvider;
    …
}

private class RemoveRuleAction extends AbstractAction {

    public RemoveRuleAction() {
        …
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        …
        firewall.updateActiveRuleSet(…);
        MainPanel mainPanel = mainPanelProvider.get();
        mainPanel.updateRulesDetailsTable();
    }
}
```

Aspect Oriented Programming

Anche per l'implementazione dell'AOP si è utilizzato Google Guice nello specifico il modulo Guice AOP che supporta before, after, and around Advice sotto forma di Interceptors di metodi. Guice AOP utilizza l'intercettazione dei metodi piuttosto che la tessitura del bytecode, che in alcuni casi può essere più semplice e veloce. L'integrazione con altre funzionalità di Guice, come l'iniezione di dipendenza e gli scope, è inoltre molto semplice. Seguendo la guida ufficiale di Guice quello che si è fatto è stato anzitutto creare una annotazione custom come segue:

```
@BindingAnnotation
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface LogPacket {

}
```

Poi si è proceduto ad applicarla a quei metodi che necessitavano dell'intervento dell'interceptor. Quindi, si è proceduto definendo l'interceptor che implementa l'interfaccia `MethodInterceptor`, come segue:

```
@Singleton
public class PacketLoggerInterceptor implements MethodInterceptor {

    @Inject
    private IPacketLogger logger;

    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
        Object result = invocation.proceed();
        if (invocation.getMethod().getReturnType() == boolean.class ||
            invocation.getMethod().getReturnType() == Boolean.class) {

            boolean allowed = (Boolean) result;
            IPacket packet = (IPacket) invocation.getArguments()[0];
            this.logger.logPacket(packet, allowed);
        }
        return result;
    }
}
```

È interessante notare come questo sia l'unico caso in cui non avvenga l'iniezione delle dipendenze tramite Costructor Injection ma tramite Property Injection e questo è dovuto al fatto che l'interceptor come si vedrà tra poco non viene gestito da Guice quindi viene creato a mano, e il

costruttore non essendo chiamato da Guice non premetterebbe l'iniezione delle dipendenze. Infine l'interceptor viene creato, viene richiesta l'iniezione delle sue dipendenze e viene legato a uno specifico Matcher (che nella nomenclatura della AOP corrisponde a un Join Point) in un modulo Guice come fatto di seguito:

```
    @Override
    protected void configure() {
        …
        PacketLoggerInterceptor interceptor =
                                                new PacketLoggerInterceptor();
        this.requestInjection(interceptor);
        this.bindInterceptor(Matchers.any(),
                             Matchers.annotatedWith(LogPacket.class),
                             interceptor);
    }
```

L'API di intercettazione dei metodi implementata da Guice fa parte di una specifica pubblica chiamata AOP Alliance. Ciò rende possibile l'uso degli stessi intercettori in diversi framework.

Thread e Sincronizzazione

Il componente ClientSimulator ha il compito di emulare un numero configurabile di client che, a intervalli regolari, inviano pacchetti al firewall. Per ottenere una simulazione realistica e scalabile la classe adotta un'architettura concorrente basata su `ExecutorService`, strumenti di sincronizzazione della `java.util.concurrent` e poche, mirate, precauzioni di thread-safety. Per orchestrare i client simulati, ClientSimulator crea un `ScheduledExecutorService` tramite `Executors.newScheduledThreadPool(…)`. L'esecutore genera un pool di thread della dimensione esatta dei client da emulare, evitando overhead di creazione dinamica, questo permette di pianificare task con periodicità fissa attraverso `scheduleAtFixedRate()`. Ogni ciclo del for (da 0 a clientsCount – 1) registra nel pool un nuovo SimulatedClientTask. Il task parte immediatamente (ritardo 0) e viene poi ripetuto ogni intervalMs millisecondi; all'interno, l'oggetto invia pacchetti fino a raggiungere maxPackets, dopo di che invoca `latch.countDown()` per segnalare la propria conclusione. Per sapere quando tutti i client hanno terminato, la classe utilizza un `CountDownLatch` inizializzato al numero di client. Un thread ausiliario, creato con new Thread(...).start(), attende bloccante su `latch.await()`. Quando il contatore raggiunge lo zero:

- Shutdown del pool – `scheduler.shutdown()` ferma la pianificazione di nuovi task e consente una dismissione ordinata;
- Aggiornamento dello stato condiviso – Il campo running (contrassegnato come volatile, quindi immediatamente visibile a tutti i thread) passa a false;
- Notifica di completamento – Viene invocato `onSimulationFinished()` su ogni listener registrato.

In questo modo il metodo pubblico `startSimulation()` ritorna subito controllo al chiamante, ma la chiusura avviene in maniera asincrona e non blocca l'interfaccia grafica che ha avviato la simulazione tramite un controller. Quando descritto in precedenza nella pratica è implementato tramite il seguente snippet di codice:

```
public void startSimulation() {
    this.running = true;
    this.listeners.
    forEach(ISimulationStatusListener::onSimulationStarted);

    ScheduledExecutorService scheduler =
                    Executors.newScheduledThreadPool(this.clientsCount);
    CountDownLatch latch = new CountDownLatch(this.clientsCount);

    for (int i = 0; i < this.clientsCount; i++) {
        scheduler.scheduleAtFixedRate(
            new SimulatedClientTask("Client-" + i, this.maxPackets,
                                    latch, this.firewall),
            0,
            this.intervalMs,
            TimeUnit.MILLISECONDS
        );
    }

    new Thread(() -> {
        try {
            latch.await();
            scheduler.shutdown();
            this.running = false;
            this.listeners.
            forEach(ISimulationStatusListener::onSimulationFinished);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }).start();
}
```

Volendo fare un approfondimento sulla Thread-safety e sulla visibilità dei dati:

- `CopyOnWriteArrayList`

  Il contenitore dei listener garantisce iterazioni sicure anche in presenza di registrazioni simultanee: le modifiche producono una copia interna, evitando `ConcurrentModificationException` senza dover ricorrere a blocchi synchronized.

- `volatile boolean running`

  Il flag di stato viene letto potenzialmente da thread diversi rispetto a quelli che lo aggiornano; il modificatore volatile assicura che i

valori scritti siano immediatamente visibili e che non vengano riorganizzati dal compilatore o dalla CPU.

- Assenza esplicita di lock

  Grazie alle strutture concorrenti e a `CountDownLatch`, non sono necessari blocchi fatti a mano, questo riduce il rischio di deadlock e semplifica l'analisi del codice.

Le primitive utilizzate in ClientSimulator rappresentano un approccio moderno alla concorrenza in Java, più robusto, leggibile e manutenibile rispetto al modello classico basato su `Thread, Runnable, wait() e notify()`.

Per garantire la Thread-Safety dei metodi del firewall che quindi possono essere chiamati in maniera concorrente da più thread si è utilizzato un Read-Write Lock tramite l'implementazione `ReentrantReadWriteLock`. Questo tipo di lock è molto utile quando si ha una risorsa condivisa che può essere letta da più thread contemporaneamente, ma scritta da un solo thread alla volta, in esclusiva. L'implementazione `ReentrantReadWriteLock` fornisce un lock rientrante ovvero lo stesso thread può acquisire più volte lo stesso lock (utile nei metodi ricorsivi o annidati). Di conseguenza ogni metodo della classe InMemoryFirewallEngine prima di effettuare una qualsiasi operazione richiede il lock più adatto come segue:

```
@Override
public IRuleSet getActiveRuleSet() {
    this.rwLock.readLock().lock();
    try {
        return this.activeRuleSet;
    } finally {
        this.rwLock.readLock().unlock();
    }
}
```

È interessante sottolineare il fatto che siccome in InMemoryFirewallEngine avviene la sincronizzazione degli accessi ai RuleSet di conseguenza in RuleSet per memorizzare le Rule si è utilizzata una collezione normale e non una più costosa collezione concorrente, nello specifico si è utilizzata una normalissima ArrayList.

Clonazione

Per gli scopi del progetto era necessario clonare l'`activeRuleSet` creando quindi il `clonedRuleSet` ovvero uno snapshot isolato del set di regole, modificabile dalla GUI senza influenzare in alcun modo l'activeRuleSet. Quindi il clonedRuleSet altro non è che una deep copy dell'activeRuleSet. Per ottenere una deep copy sono stati necessari diversi passi:

- La classe RuleSet implementa `Clonable` e quindi sovrascrive `clone();`
- All'interno del metodo clone anzitutto viene fatta una shallow copy della base class, poi viene sostituita la lista delle regole con una nuova lista vuota, infine si itera sulla lista delle regole e viene aggiunto il clone di ciascuna regola alla lista vuota del clone del RuleSet;
- Anche la classe Rule implementa `Clonable` e quindi sovrascrive `clone();`
- All'interno del metodo clone viene fatta una shallow copy della base class, e poi tutti i campi della Rule vengono settati normalmente nel caso di campi String (poiché immutabili grazie al Pattern Flyweight), viene invece chiamato il metodo clone per i campi aventi tipo non primitivo come PortRange e IPRange;
- Anche le classi PortRange e IPRange implementano `Clonable` e quindi sovrascrivono `clone();`
- Nel caso di PortRange il metodo clone è standard avendo esso solo campi primitivi;
- Nel caso di IPRange il metodo clone, dopo aver fatto una shallow copy della base class, clona un campo non-clonabile appartenente a una libreria esterna ricreandolo via costruttore.

Non viene allegato alcuno snippet di codice relativo all'implementazione di quanto descritto in precedenza per motivi di spazio, fare riferimento al codice sorgente dell'applicativo.

Bibliografia

I. Provider<T> Guice Documentation: https://google.github.io/guice/api-docs/3.0/javadoc/com/google/inject/Provider.html
II. Aspect Oriented Programming in Guice: https://github.com/google/guice/wiki/AOP
III. AOP Alliance (Java/J2EE AOP standards). https://aopalliance.sourceforge.net/