ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Sistemi Distribuiti

# MOK MIDDLEWARE: NETWORKING FRAMEWORKS

*Relazione di*:
GIACOMO DRADI
*Matricola*:
726868

ANNO ACCADEMICO 2014-2015

# Indice

# 1. Introduzione

Questa relazione ha lo scopo di documentare il più dettagliatamente possibile l'attività progettuale prevista nel corso di Sistemi Distribuiti 2014/2015. Tale attività prevede di comprendere, analizzare, e possibilmente dare un contributo attivo al progetto **MoK - Molecules of Knowledge** creato dai prof. Andrea Omicini e Stefano Mariani, e sul quale già da alcuni anni lavorano.

## 1.1 Goals

Gli obiettivi principali che si intendono raggiungere grazie alla partecipazione al progetto sono:

- comprendere aspetti avanzati di coordinazione, sistemi multi agente (MAS) e middleware;
- lavorare in ambiti di ricerca attuali utilizzando tecnologie già largamente diffuse;
- contribuire attivamente allo sviluppo di nuove tecnologie e soluzioni;

L'intero progetto MoK è stato suddiviso in quattro macro aree, ciascuna a sua volta suddivisa in sub-projects assegnati ai vari studenti. Grazie a questa ripartizione è possibile e auspicabile che il lavoro prodotto dai vari gruppi possa essere adeguatamente integrato per estendere quanto già presente.

## 1.2 MoK

Molecules of Knowledge è un modello di coordinazione che promuove la self-organisation della conoscenza in un sistema multi agente (MAS) [Omicini, 2011]. Il contesto in cui esso si inserisce sono i Knowledge-Intensive Environments (KIE), cioè sistemi che combinano processi applicativi, tecnologie e capacità delle persone per *gestire* grandi depositi di informazioni, tipicamente non strutturati (blog, pagine web, ecc). Alcune caratteristiche principali dei KIE sono:

- grandi quantità di dati (TB);
- necessità di scalabilità;

- diversità sia nella rappresentazione dell'informazione che nella destinazione d'uso;
- openness nei confronti di nuovi utenti del sistema;

Proprio da queste necessità, a cui un aumento del potere computazionale non potrà dare risposta per sempre, nasce l'idea di fondo di MoK:

*Data is alive, information is a living thing continuously and spontaneously interacting with other information as well as with its prosumers, evolving itself according to such interactions* [Mariani, 2011].

Un sistema MoK può essere visto come una rete di repository di informazioni, in cui delle sorgenti continuamente e spontaneamente inseriscono frammenti di dati. Questi frammenti possono poi aggregarsi a seconda di specifici knowledge-related patterns o diffondersi all'interno della rete per raggiungere gli utenti potenzialmente interessati. Gli utenti che interagiscono con questi dati lasceranno nel sistema delle tracce che saranno poi utilizzate dal sistema stesso per influenzare l'evoluzione della conoscenza in modo trasparente all'utente.

### 1.2.1 MoK Model

Allo stato attuale, il modello di Molecules of Knowledge prevede i seguenti componenti:

- **semi:** le sorgenti di informazioni; continuamente e spontaneamente inseriscono atomi nei compartimenti;
- **compartimenti:** sono i repository attivi di informazioni, che memorizzano semi, atomi, molecole e enzimi, e sono responsabili della pianificazione ed esecuzione di reazioni;
- **catalizzatori:** gli utenti del sistema MoK, che manipolano le informazioni influenzando il sistema;
- **atomi:** le più piccole unità di informazione in MoK; contengono un identificatore del loro seme generatore, il contenuto (l'informazione), metadati per aiutare l'intepretazione semantica e il valore della concentrazione dell'atomo.
- **molecole:** aggregazioni di atomi prodotte continuamente e spontaneamente dai compartimenti; gli atomi aggregati devono avere contenuto semanticamente correlato;
- **enzimi:** tracce lasciate automaticamente dagli utilizzatori del sistema che vengono utilizzate per veicolare le reazioni.

Gli atomi si riuniscono in molecole attraverso quattro reazioni, che rappresentano le regole di coordinazione secondo il modello biochimico:

- **aggregazione:** accorpamento di atomi-molecole semanticamente correlati;

- **diffusione:** movimento di molecole tra compartimenti vicini;
- **rinforzo:** aumento della concentrazione attraverso il consumo di enzimi;
- **decadimento:** diminuzione della concetrazione con il passare del tempo;

### 1.2.2 Case study

Al giorno d'oggi non sono presenti sistemi MoK o simulazioni in larga scala. Un prototipo di MoK-Engine è stato tuttavia realizzato sull'infrastruttura TuCSoN e alcune simulazioni del comportamento di self-organisation sono state effettuate utilizzando il Bio-PEPA framework.

**MoK-News** è un modello realmente implementato di MoK pensato per la gestione di notizie, uno scenario caraterizzato da eterogeneità, ubiquità e non predittività, caratteristiche che lo rendono particolarmente adatto all'applicazione di MoK. MoK-News è stato realizzato su TuCSoN, una piattaforma per la coordinazione distribuita tra sistemi multi agente. A ogni tag presente in NewsML e NITF, i due principali standard per la rappresentazione di notizie secondo IPTC (International Press Telecommunications Council), è stato mappato un atomo MoK. A ogni utente del sistema (giornalisti, ecc) è assegnato un proprio compartimento (ad esempio tramite smartphone) in cui eseguire le operazioni tramite apposite GUI. All'interno del compartimento sorgenti, atomi ed enzimi interagiscono continuamente e autonomamente per offrire il miglior supporto possibile all'utente.

# 2. Overview

Il progetto MoK è stato suddiviso in quattro ambiti di ricerca, che coicidono con i quattro principali componenti logici della tecnologia. Per ogni ambito di ricerca sono stati definiti diversi progetti che potranno poi essere integrati per costituire un unico ecosistema MoK. Ogni progetto è sufficientemente modulare per permettere separate attività e prodotti software. I quattro ambiti di ricerca sono:

1. Information exploration and exploitation: leveraging Behavioural Implicit Communication in distributed shared workspaces;
2. Distributed information harvesting: from unstructured text to semantically enriched MoK atoms;
3. Information persistency: scalable distributed storage solutions (e.g. key-value DBs, graph DBs, document-oriented DBs);
4. Information management: decentralized self-* algorithms, semantic reasoning on resource-constrained devices, efficient in-memory DBs.

## 2.1 MoK middleware: networking frameworks

Il progetto a me assegnato si inserisce all'interno dell'area *Information management: decentralized self-* algorithms, semantic reasoning on resource-constrained devices, efficient in-memory DBs.* Scopo del progetto è trovare, analizzare e confrontare diversi framework di comunicazione che soddisfino il più possibile determinati requisiti imposti. Scelto il framework piú adatto, esso verrá utilizzato per implementare un *networking framework* che possa essere integrato in MoK attraverso opportune API.

### 2.1.1 Requisiti

Alla luce dei requisiti forniti, il framework scelto deve necessariamente:

1. essere sviluppato in Java;
2. essere utilizzabile su dispositivi mobile Android.

Fattori di merito del framework sono:

1. basso uso di risorse (efficienza);
2. alta frequenza di messaggi scambiati (performance);
3. supporto a multi-protocollo (nativamente o tramite API);
4. supporto a diverse modalità di comunicazione (point-to-point, broadcast, ecc).

I framework suggeriti da cui partire la ricerca sono:

- Netty: (asynchronous event-driven network application framework);
- Kryonet: UDP/TCP network communication framework, for efficient sharing of object graphs.

## 2.2 Netty

Netty è un framework NIO (Non-blocking input/output) per client e server che permette un facile e veloce sviluppo di applicazioni network [7]. Le caratteristiche principali sono:

- API unificate per socket bloccanti e non bloccanti.
- Permette la *separation of concerns* attraverso un modello ad eventi flessibile ed estensibile;
- Gestione dei thread personalizzabile (single-thread, thread pool, ecc).

Netty 4.0, l'ultima stable-release, è scritta in Java e supporta Android ufficialmente [10]. I protocolli forniti nativamente sono TCP, UDP, UDT, SCTP, ma è possibile realizzare protocolli custom sopra di essi in modo rapido. Netty fornisce codec di supporto per i protocolli:

- HTTP 1.0, HTTP 2.0;
- WebSocket (+ compressione);
- Protobufs, JBoss Marshalling, Java Serialization
- DNS;
- Proxy;
- SSL/TLS.

è inoltre possibile servire diversi protocolli sulla stessa porta TCP/IP.

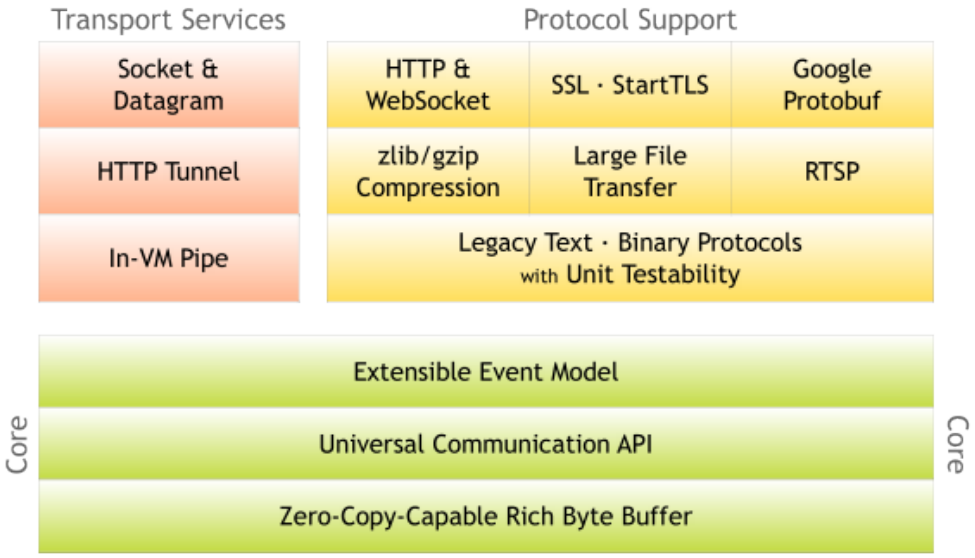


Figura 1: Architettura di Netty.

### 2.2.1 Modello

Netty è completamente asincrono, poichè sfrutta java.nio per realizzare input/output non bloccante. È possibile un approccio ad eventi grazie all'utilizzo di *callback*, *future* e *promise*. Il modello utilizzato prevede i seguenti componenti principali [8]:

- Channel: componente in grado di realizzare la comunicazione. Permette di:
  - conoscere lo stato del canale (aperto, connesso, ecc);
  - conoscere parametri di configurazione (dimensione del buffer ricevente, ecc);
  - effettuare operazioni di I/O (read, write, connect, bind);
- ChannelHandler: intercettatori di eventi/operazioni su un channel.
- ChannelPipeline: lista di ChannelHandler che processano eventi in cascata.
- EventLoop: gestisce le operazioni dei vari Channel.

Ogni channel viene registrato a un EventLoop (1 per thread) e tutti gli eventi sono processati nello stesso thread. Ogni EventLoop serve molti Channel. I cambiamenti dello stato/eventi del channel vengono intercettati da ChannelHandler, secondo il pattern *interceptor*, garantendo flessibilità a seconda del bisogno. Catene di ChannelHandler in ChannelPipeline permettono di realizzare logica complessa in modo modulare.

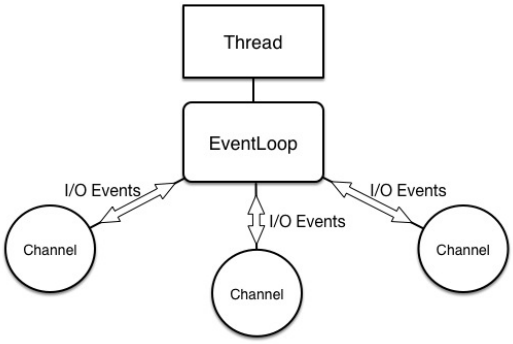


Figura 2: EventLoop in Netty [12]

I codec forniti da Netty per supportare i vari protocolli sono implementazioni di ChannelHandler per ogni specifico protocollo, che possono essere utilizzate direttamente, concatenate in ChannelPipeline o estese per realizzzare le funzionalità richieste. Ad esempio, i due ChannelHandler:

1. *io.netty.handler.codec.serialization.ObjectEncoder*
2. *io.netty.handler.codec.serialization.ObjectDecoder*

sono codec che permettono di realizzare lo scambio di oggetti Java (POJO) in modo estremamente veloce: è sufficiente aggiungere entrambi alla ChannelPipeline.

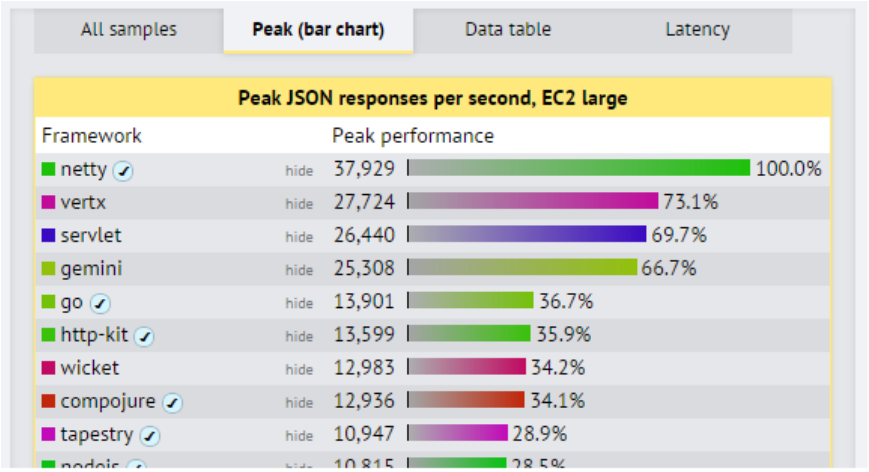


Figura 3: Performance di Netty in JSON/s [3]

### 2.2.2 Performance

La natura NIO di Netty permette un minor uso di risorse necessarie rispetto a soluzioni non-NIO sopratutto in caso di un elevato numero di connessioni. Test di benchmark sono stati effettuati per verificare le performance di vari framework; nel test valutante il numero di risposte JSON al secondo, di un oggetto istanziato con una semplice stringa, Netty si è posizionato al primo posto.

## 2.3 KryoNet

KryoNet è una libreria Java che fornisce API semplici e pulite per realizzare comunicazione tra client e server utilizzando TCP e UDP. Come Netty, anche KryoNet utilizza input/output non bloccante (NIO). KryoNet è particolarmente efficace nello scambio di object graphs, grazie all'utilizzo di una propria libreria di serializzazione [6].

KryoNet è compatibile con Android.

KryoNet mette a disposizione nativamente due tipi di comuncazioni:

1. Scambio di oggetti Java
2. RMI *(Remote Method Invocation)*

Forme più complesse di interazione devono essere implementate manualmente al di sopra di questi. RMI aggiunge meno overhead rispetto all'invio di un intero oggetto, e si occupa del marshaling e delle operazioni di basso livello. Tuttavia è necessario che chi utilizza metodi chiamati tramite RMI gestisca manualmente errori o attese nella comunicazione remota.

KryoNet utilizza TCP. È possibile utilizzare opzionalmente UDP in aggiunta a TCP, ma non è suggerito utilizzare entrambi contemporaneamente, in quanto grosse quantità di dati potrebbero interferire tra le due porte. KryoNet non fornisce nessun supporto particolare alla gestione delle problematiche di UDP (es: gestione del flusso dei pacchetti); se necessari, esse devono essere gestite manualmente.

Poichè KryoNet ha come scenario principale applicazioni client-server tipiche dell'online gaming, ha un insieme di API pulite ed eleganti, ma limitate all'ambito specifico per cui è stato pensato.

## 2.4 Alternative

Oltre ai due framework suggeriti, ho compiuto varie ricerche allo scopo di trovare eventuali altri framework compatibili con i requisiti presentati. Sono stati trovati:

- Apache MINA [1]

- Grizzly [4]

Altre alternative, come CoralReactor [2] sono state scartate per il costo di setup.

### 2.4.1 Apache MINA

Apache MINA ùn framework di comunicazione sviluppato da Apache. Poichè Netty è nato partendo da MINA, e cercando di risolverne le *issues* ricorrenti, MINA si è rilevato come una versione di Netty ma con API di più basso livello, e generalmente di difficoltà maggiore [9]. Ad esempio, il concetto di ChannelPipeline in Netty, molto utile per realizzare comportamenti o protocolli custom, in MINA richiede una implementazione ad hoc.

### 2.4.2 Grizzly

L'obiettivo di Grizzly è aiutare gli sviluppatore nella realizzazione di server robusti e scalabili sfruttando Java NIO, e offrire una serie di componenti per gestire funzionalità aggiuntive come SSL, WebSocket, Comet, ecc). Grizzly quindi si concentra sopratutto lato server, e in particolare ciò che riguarda la tecnologie Web (WebSocket, long-polling tramite Comet, WebService). Le caratteristiche principali di Grizzly sono [5]:

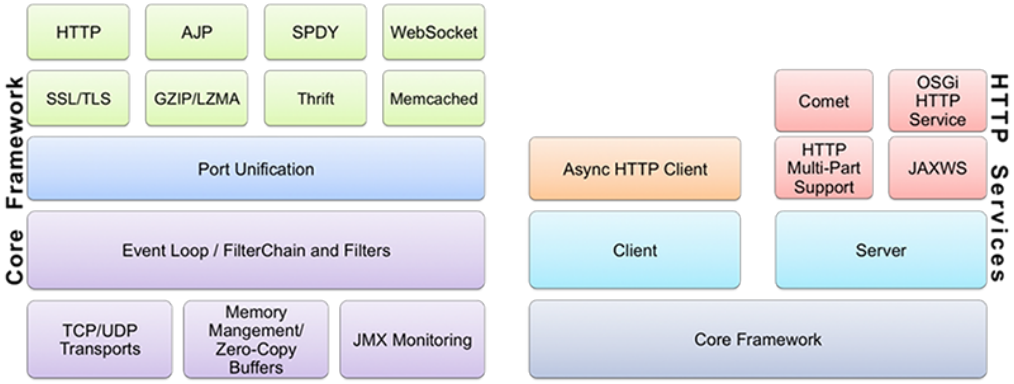


Figura 4: Stack di Grizzly

- utilizzo di socket non bloccanti;
- gestione del buffering ottimizzata;
- gestione customizzabile dei thread (thread-pool, ecc)
- supporto alla unificazione porte;
- framework HTTP completo;

Grizzly presenta concetti simili a Netty:

- Transport: rappresenta l'implementazione di un trasporto di una rete;
- Connection: è un traposrto connesso (l'equivalente di un Channel in Netty);
- Filter: rappresenta un task da compiere sulla connessione (simile al ChannelHandler);
- FilterChain: una serie di filtri (ChannelPipeline in Netty).

Grizzly predispone un ampia gamma di componenti per la gestione avanzata di tecnologie Web in modalità NIO, che generalmente è il caso d'uso principale di Grizzly.

## 2.5 Conclusioni

Dopo una fase di analisi, ho identificato alcuni parametri e, per ogni framework, ho assegnato una valutazione. Per ciò che riguarda benchmark e analisi delle performance in larga scala, non sono state trovate informazioni o articoli sull'argomento; Twitter ha però annunciato che il passaggio a Netty ha portato a un miglioramento del tempo di risposta a una ricerca di tre volte [11]. Alla luce di quanto espresso, ritengo che il framework più

| | **Netty** | **KryoNet** | **MINA** | **Grizzly** |
|---|---|---|---|---|
| *Supporto Android* | UFFICIALE | SI | SI | SI |
| *TCP/UDP* | SI | SI | SI | SI |
| *HTTP* | CODEC | NO | CODEC | SI |
| *WebSocket* | CODEC | NO | CODEC | SI |
| *SSL* | SI | NO | SI | SI |
| *Flessibilità* | OTTIMA | SCARSA | BUONA | OTTIMA |
| *Semplicità* | BUONA | BUONA | SCARSA | SCARSA |
| *Documentazione* | OTTIMA | SCARSA | SCARSA | BUONA |

Figura 5: Comparazione tra framework

conveniente per il task assegnato sia Netty. Esso soddisfa i requisiti richiesti, ha un ampia documentazione, implementa nativamente molti protocolli e fornisce strumenti per l'aggiunta di nuovi. È sufficientemente flessibile, ha ottime performance ed è compatibile

con Android.

Lo scambio di object graphs è una funzionalità particolarmente utile per MoK. Le MoK-Molecules sono un esempio di oggetto che ha necessità di poter essere scambiato tra vari compartimenti, rappresentati dai device connessi attraverso il framework. Netty, grazie ai codec di serializzazione e deserializzazione forniti, può soddisfare questa necessità. Qualora fossero necessarie particolari implementazioni della serializzazione, sarebbe sufficiente estendere il codec fornito.

# Project

## 3.1 Overview

Il primo passo svolto è stato trovare quali funzionalità dovesse avere il networking framework, attraverso la lettura di vari paper, sulla base delle necessità di MoK. È risultato che il livello di astrazione necessario era, come si poteva intuire, di più alto livello rispetto a quello previsto in Netty. Per questo motivo ho realizzato un framework, costruito al di sopra di Netty, che fornisse delle API opportune per MoK.

La caratteristica principale del modello di MoK che é stata decisa supportare, e per cui si rende necessaria la presenza di un networking framework, è la reazione di diffusione delle molecole tra compartimenti. Di seguito riporto, tramite l'ausilio di diagrammi UML, la modellazione del dominio.

## 3.2 Modello

L'interfaccia principale fornita dal framework è rappresentata dal *LocalCompartment*, astrazione del compartimento di MoK. Ogni utilizzatore del framework deve, come prima operazione, istanziare un LocalCompartment, specificando le proprietà possedute da quel compartimento. Poichè tali proprietà sono specifiche di ogni ambito in cui MoK puó essere utilizzato, esse sono liberamente estendibili e customizzabili.

La caratteristica principale di ogni compartimento è la capacità di diffondere molecole ad altri compartimenti. Nella modellazione scelta, i compartimenti sono raggruppati in vicinati (*Neighbourhoods*) e la diffusione è possibile solo all'interno del proprio vicinato. Nel vicinato, ogni compartimento è associato ai propri vicini tramite delle *membrane*, che permettono lo scambio di molecole. Le membrane sono dotate di proprietà, anch'esse personalizzabili, che influenzano come o quando lo scambio deve avvenire.

Poiché l'effettiva implementazione di una molecola dipende dal modello generale di MoK, e non dal networking framework, é stata supposta l'esistenza di una generica IMolecule, che sará fornita dall'implementazione di MoK, che verrá incapsulata in DiffusingMolecule e DiffusingMolecules dal framework.


Figura 6: Struttura di un LocalCompartment

L'utilizzatore del framework, dopo aver istanziato il proprio LocalCompartment, deve richiamare la funzione JoinMoK. In questo modo riceve il proprio vicinato, costituito dalla lista di membrane create, ottenibile tramite getNeighbourhood(). Utilizzando tali membrane é possibile richiamare i vari metodi per la diffusione (singlecast, multicast o broadcast rispetto al vicinato) di una o piú molecole.

La proprietá presente nativamente nel framework, indipendente dalla particolare ap-


Figura 7: Vicinato, membrana e generica proprietá

plicazione di MoK, é il concetto di rate di diffusione, rappresentato come un delay tra l'ordine di diffusione e la sua effettiva applicazione. Nel framework é infatti possibile specificare, per ogni membrana, la distanza tra i due compartimenti che collega utilizzando le proprietá di entrambi. Maggiore é questa distanza, maggiore sará il delay di diffusione. Ogni membrana puó avere anche altre proprietá, dipendenti dal particolare scenario applicativo, che possono essere utilizzate.

Le molecole che il framework diffonde sono modellate in *DiffusingMolecole*(singola molecola) e *DiffusingMolecules* (insieme di molecole). Esse aggiungono alla molecola di MoK un identificatore, che permette al framework di conoscere univocamente quali molecole sono state diffuse e quali no.

Il framework si basa sul modello ad eventi. Le API fornite dal LocalCompartment sono non bloccanti (mantenendo il comportamento di Netty), e terminano immediatamente. Per essere notificati del risultato delle operazioni effettuate (diffusioni, join) é possibile registrare uno o piú *CompartmentObserver* le cui callback verranno richiamate alla terminazione della operazione a cui si riferiscono. Attraverso i parametri delle callback di diffusione é sempre possibile risalire a quale membrana e a quale molecola

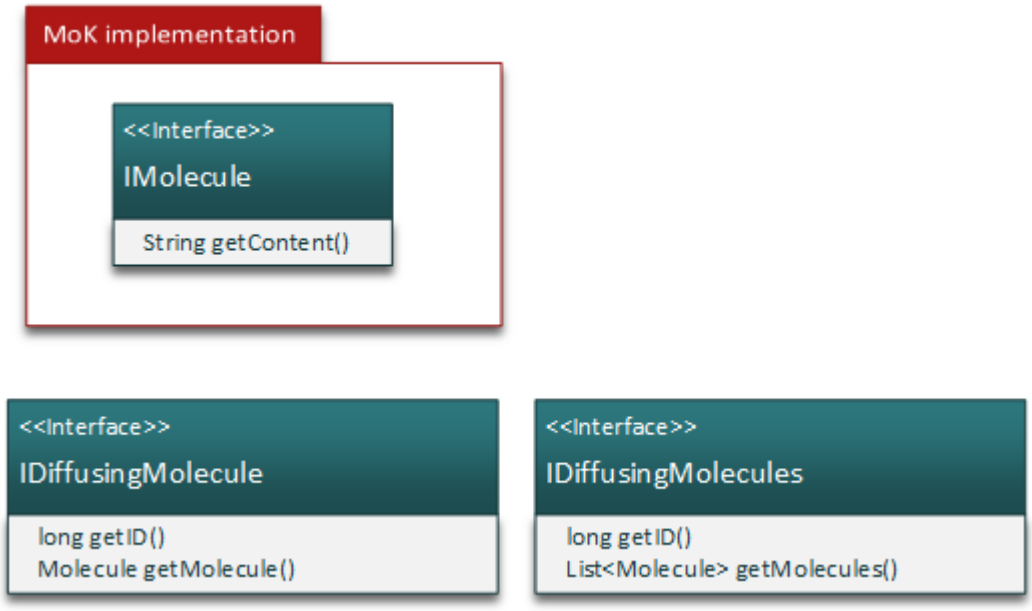


Figura 8: Modello delle molecole

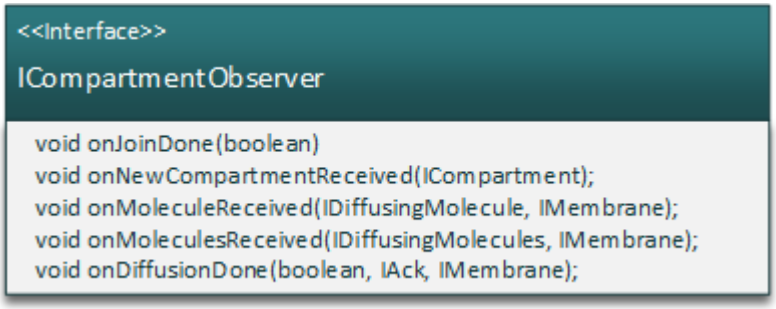


Figura 9: Observer di un LocalCompartment

(tramite ID) la callback si riferisce.
Il CompartmentObserver rileva anche l'aggiunta di un nuovo compartimento al vicinato e la ricezione di una molecola diffusa da un vicino. É possibile estendere le funzionalitá di base di un CompartmentObserver a seconda delle particolari esigenze: si potrebbe creare un MembraneObserver che, specificata una membrana, si attivi solamente quando l'operazione di diffusione riguardi tale membrana.

Qualora si volesse passare ad un modello asincrono, l'implementazione attuale consentirebbe un refactory molto veloce: sarebbe sufficiente far memorizzare i vari eventi (segnalati dalle callback) in una lista, alla quale sarebbe poi possibile accedere in modo totalmente asincrono.

## 3.3 Implementazione

Il modello é stato concretizzato in un prototipo funzionante su protocollo TCP/IP, ma in modo da poter essere successivamente esteso ad altri protocolli su altre infrastrutture.

### 3.3.1 Topologia di rete

Ogni LocalCompartment di MoK deve essere in grado di comunicare con gli altri compartimenti utilizzando la rete. Per poterne conoscere l'indirizzo, sono solitamente disponibili tre alternative:

1. approccio full peer-to-peer, in cui ogni peer invia in broadcast dei pacchetti di discovery per verificare la presenza di altri peer, e ne verifica personalmente l'appartenenza al proprio vicinato.
2. approccio centralizzato client-server, dove tutti i client (i LocalCompartment) si collegano a un server well-known che si occupa di stabilire i vicinati e restituire ai client la lista dei vicini giá pronta.
3. approccio ibrido, in cui sono presenti piú server well-known a cui i client si connettono. Ogni server well-known (*hotspot*) restituisce ai client una porzione del vicinato, quella di sua competenza, e ciascun client compone il proprio vicinato unendo i dati ricevuti dai vari hotspot.

La scelta adottata é stata la terza, l'approccio ibrido: ogni LocalCompartment mantiene una lista degli indirizzi degli hotspot, e in fase di join si connette ad ognuno parallelamente. É facile notare come, specificando un solo hotspot, si possa passare immediatamente all'approccio centralizzato. In questo approccio, una volta che ogni client ha ottenuto gli indirizzi dei vicini, si connette e comunica direttamente con loro senza la mediazione del server.

### 3.3.2 Comunicazione

Per mantenere generalitá, sono stati pensati due componenti che si occupino della comunicazione fisica tra LocalCompartment vicini e tra LocalCompartment e Hotspot: *PeerHandler* e *HotspotHandler*. Essi vengono inizializzati e utilizzati dal LocalCompartment, il quale delega a loro l'effettivo compimento delle operazioni a lui richieste. Qualora si volesse utilizzare un protocollo diverso da quello giá implementato (TCP/IP), sarebbe sufficiente creare nuove implementazioni di questi componenti, rendendo il tutto

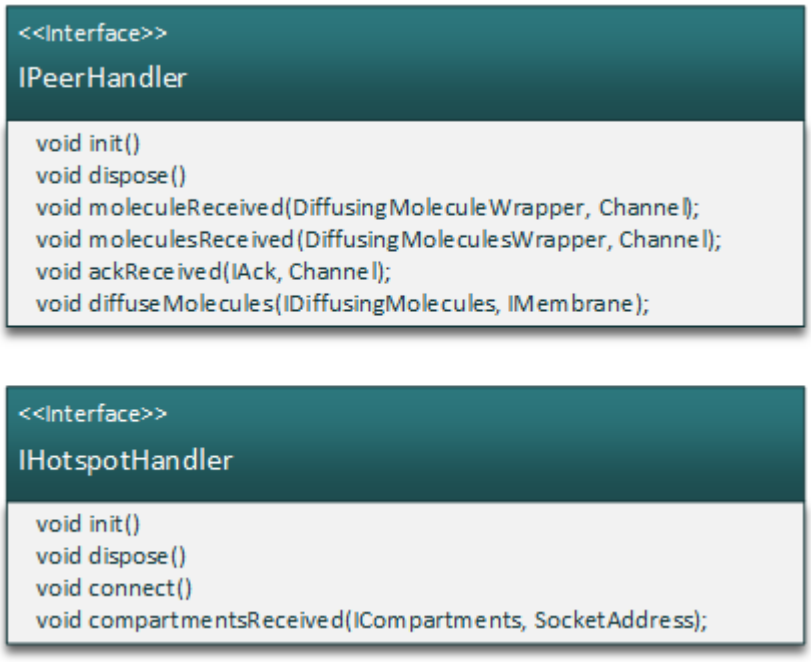


Figura 10: PeerHandler e HotspotHandler

trasparente al LocalCompartment.

Nell'implementazione corrente, la comunicazione con il server avviene in questo modo:

1. alla chiamata di joinMoK, un HotspotHandler viene istanziato.
2. l'HotspotHandler si connette ad ogni hotspot e invia ad ognuno il proprio Compartment con le sue proprietá.
3. se nessuna connessione é andata a buon fine, o se il timeout di connessione scade, la callback di join terminata viene chiamata con risultato negativo.
4. se almeno 1 connessione é stata stabilita, si attendono le parziali liste di Compartment da tutti gli hotspot connessi e si richiama la callback positivamente.

I dati vengono scambiati utilizzando il formato JSON. Ottenuta la lista di vicini, si creano le rispettive membrane che saranno memorizzate nel Neighbourhood, accessibile dal LocalCompartment. Le connessioni con gli hotspot vengono chiuse e l'HotspotHandler fermato. Solo dopo che la callback di join é stata lanciata positivamente é possibile effettuare le operazioni di diffusione o di richiesta del vicinato, che non avrebbero alcun effetto altrimenti.

La comunicazione tra Compartment avviene in questo modo:

1. viene mantenuta una HashMap di connessioni attive tra Compartment (equivalente delle membrane), inizialmente vuota.
2. alla richiesta di una nuova diffusione su una membrana, si controlla se esiste giá una connessione per quella membrana. Se esiste, si seleziona quella, altrimenti se ne stabilisce una nuova e si memorizza nella HashMap.
3. Viene aggiunto l'ID del LocalCompartment mittente alla molecola o alla lista di molecole da inviare e si procede all'invio utilizzando la connessione precedentemente selezionata.

Anche in questa comunicazione i dati scambiati sono in formato JSON. La diffusione viene confermata attraverso l'invio di ACK contenenti l'ID della molecola (o della lista) scambiata. In questo modo é possibile per il mittente sapere quando una diffusione ha avuto successo, e notificarlo tramite callback.

Un PeerHandler rileva anche la ricezione di molecole o di compartimenti. Nel primo caso, viene chiamata la callback di molecola (o lista di molecole) ricevuta che contiene anche l'informazione sulla membrana utilizzata. Il secondo caso si presenta invece quando un nuovo compartimento si é connesso al sistema, gli hotspot hanno deciso che apparteneva a un vicinato giá popolato e hanno inviato ai compartimenti di quel vicinato le informazioni circa il loro nuovo vicino.

### 3.3.2 Hotspots

Come anticipato, l'approccio ibrido scelto prevede l'esistenza di server well-known, chiamati *Hotspot*, a cui i LocalCompartment si connettono inizialmente per ottenere le informazioni circa il proprio vicinato, e che notifichino successivamente l'aggiunta di un nuovo vicino.

Poiché decidere l'appartenenza di un compartimento a un vicinato é un compito esterno al networking framework, in fase di lancio dell'Hotspot é necessario specificare il criterio con il quale decidere tale appartenenza. Nell'implementazione attuale, viene considerato un solo vicinato, che si compone tutti i Compartment connessi.

Ogni Hotspot mantiene una lista degli compartimenti che si sono connessi. Ogni Compartment ha un ID univoco. Alla connessione di un nuovo Compartment, l'Hotspot:

1. controlla se il compartimento era giá presente nella lista, nel caso lo sovrascrive, altrimenti lo aggiunge.
2. invia al compartimento connesso la lista di tutti i compartimenti gia connessi che ritiene appartengano al vicinato del nuovo connesso.
3. ad ognuno dei compartimenti inviati al nuovo connesso invia la notifica di nuova aggiunta al vicinato, con le informazioni del nuovo connesso.

## 3.4 Test case

É stato realizzato un prototipo funzionante di quanto implementato a scopo di test. Si tratta di un Compartment con GUI che permette di effettuare join e varie diffusioni con altri peer connessi. Per l'esecuzione é necessario:

1. lanciare un hotspot (eventualmente specificandone la porta nel file *hotspot.components.tcp.TCPHotspot*) attraverso il file *networking.hotspot.Launcher*;
2. creare un jar per ogni LocalCompartment da istanziare, specificando come main quello contenuto nel file *networking.peer.SimpleTest*;
3. creare una directory per ogni LocalCompartment, inserendo all'interno una copia del jar, dei file *hotspots.txt* e *properties.txt*. Assicurarsi che il file *hotspots.txt* contenga l'indirizzo dell'hotspot appena lanciato.
4. lanciare i jar tramite il comando
   *java -jar LocalCompartment.jar*

Ogni LocalCompartment crea nella propria directory un file *uuid.txt* contenente l'ID unico generato per quel compartimento.

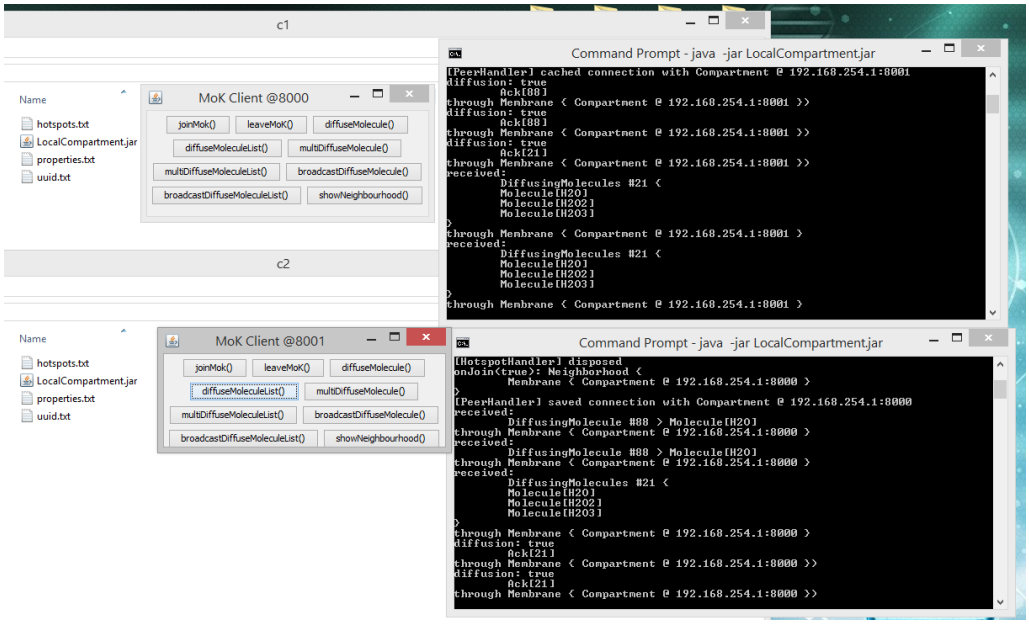


Figura 11: Esempio di test con due LocalCompartment

# Bibliografia


[1] Apache mina. https://mina.apache.org/mina-project/faq.html.

[2] Coralreactor. http://www.coralblocks.com/index.php/category/coralreactor/.

[3] Frameworks round 2. http://www.techempower.com/blog/2013/04/05/frameworks-round-2/.

[4] Grizzly. https://grizzly.java.net/.

[5] Grizzly overview. http://java.cz/dwn/1003/70612_Grizzly_JUG.pdf.

[6] Kryonet. https://github.com/EsotericSoftware/kryonet.

[7] Netty. http://netty.io/.

[8] Netty: Basics for beginners. http://bhardwaj-gaurav.blogspot.it/2014/02/netty-basics-for-beginners.html.

[9] Netty vs apache mina. http://stackoverflow.com/questions/1637752/netty-vs-apache-mina.

[10] New and noteworthy in 4.1. http://netty.io/wiki/new-and-noteworthy-in-4.1.html.

[11] Twitter and netty. https://blog.twitter.com/2011/twitter-search-now-3x-faster.

[12] Why netty? http://normanmaurer.me/presentations/2014-netflix-netty/slides.html.