MoK Project:
Communication for Pervasive Systems

Luca Santonastasi, Matteo Fattori

Marzo 2015

Capitolo 1

Il Concetto di Diffusione

Nel modello MoK, la diffusione è vista come un meccanismo di migrazione della conoscenza nel quale atomi e molecole possono condividere informazioni solamente con i compartimenti vicini. Assumendo σ come identificatore di un compartimento e $\| \cdot \|_\sigma$ come l'insieme di molecole presenti in tale compartimento , la reazione di diffusione è modellata come segue:

$$\begin{aligned} &\|Molecules_1 \cup molecule_1\|_{\sigma\prime} + \|Molecule2\|_{\sigma\prime\prime} \\ \xrightarrow{r_{diffusion}} &\|Molecules_1\|_{\sigma\prime} + \|Molecule2 \cup molecule_1\|_{\sigma\prime\prime} \end{aligned} \tag{1.1}$$

Quindi per diffusione si intende la migrazione di *Molecoles1* da un compartimento ad uno di quelli presenti nel suo vicinato. Notiamo che, la formula sopra specificata non incapsula alcuna informazione riguardo alla composizione geografica della rete o allo stato delle diverse entità presenti nel sistema, quindi possiamo presumere di dover gestire tutta una serie di scenari basandoci su fatti che potrebbero accadere durante la computazione del sistema. La comunicazione fra le diverse entità può dipendere sia dalle situazioni in cui un compartimento può trovarsi, ad esempio si necessita di informazioni che non appartengono a nessuna delle molecole presenti nel compartimento locale, sia dall'azione specifica che un ipotetico utente potrebbe voler eseguire per poter ad esempio aggiornare le informazioni. Inoltre MoK è pensato per attivare autonomamente, secondo criteri probabilistici, lo scatenamento di reazioni legate alla modifica dei compartimenti. Vi è la necessità di astrarre dal cambiamento dinamico del proprio vicinato. Questi requisiti iniziali ci hanno portato allo sviluppo di 3 casi d'uso nel quale un attore detiene un compartimento e può eseguire una serie di azioni:

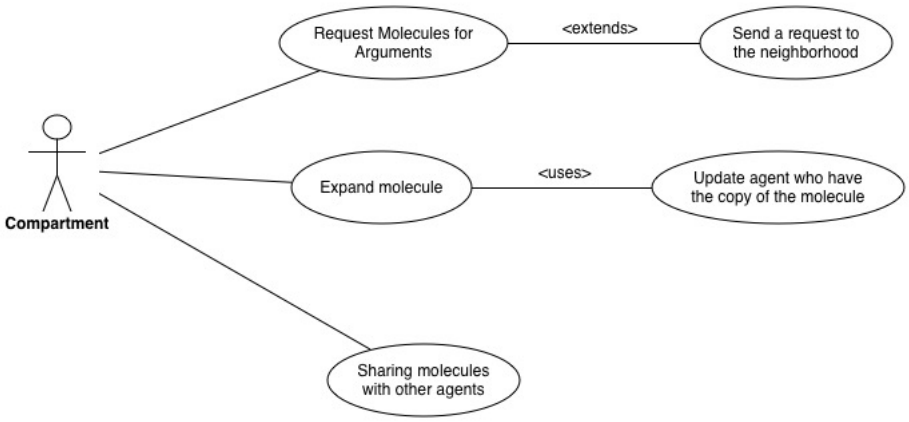


Figura 1.1: Case study

Nel primo caso d'uso abbiamo dato al nostro attore la possibilità di richiedere e recuperare informazioni, che nel caso di MoK risiedono in un compartimento sotto forma di molecole. Questo primo approccio astrae dal fatto che una molecola può risiedere nello stesso compartimento dell'attore, egli sarà del tutto ignaro delle coordinate reali della molecola.
Il secondo invece tratta la possibilità di aggiornare le informazioni presenti nel sistema e quindi, più in generale, espandere una molecola presente. Tale aggiornamento non può essere solamente locale ma deve essere condiviso con chiunque è interessato all'informazione.
Il terzo caso d'uso è relativo al fatto che, a fronte di una richiesta, il compartimento di un attore deve poter condividere le proprie informazioni sotto forma di molecole se tali informazioni sono contenute nel proprio compartimento.

Descriviamo ora come le nostre primitive possono far fronte ai casi d'uso descritti:

- Broadcast:

  Primitive 1.1.

  requestKnowledge(compartmentName,[interestedArgs])

  Tale primitiva viene utilizzata quando un attore si avvia, o vuole, recuperare conoscenza dal proprio vicinato. Per *interestedArgs* si intende una lista di argomenti ("topic", tema, categoria di un discorso ecc..) a cui l'attore, o meglio il compartimento, è interessato. Quando uno dei

compartimenti vicini riceve la richiesta, esso crea in maniera autonoma un nuovo elemento nella propria struttura dati che avrà come chiave il nome del compartimento che ha effettuato la richiesta e come valore una lista di argomenti ("topic", tema, categoria di un discorso ecc..) a cui esso è interessato. In questo modo si tiene traccia degli argomenti presenti nel vicinato e/o condivisi per facilitare le operazioni di aggiornamento.

- Unicast:

Primitive 2.1.

ackRKS(*uniqueId*,compartmentName,Argument,[Molecule])

Dopo aver ricevuto una *requestKnowledge*, il compartimento invia una risposta descritta da *ackRKS* , per ogni argomento presente nella lista della primitiva precedente. Se invece il compartimento non ha al suo interno alcuna molecola inerente a nessuno degli argomenti richiesti, allora egli non invierà nulla. Il parametro *Argument* quindi rappresenta l'argomento ("topic", tema, categoria di un discorso ecc..) di risposta e *[Molecule]* rappresenta una lista di molecole contenente le informazioni relative all'argomento specificato. Quando la risposta *ackRKS* viene ricevuta da un compartimento, questo inserirà la molecola qualora non ne abbia già ricevuta una con il medesimo *uniqueId*, ed aggiornerà la propria struttura dati interna per tener traccia degli argomenti presenti nel vicinato e/o condivisi. Grazie al parametro *uniqueId* preveniamo quindi il problema di inserimenti multipli della stessa molecola all'interno di un compartimento, qualora più di uno dei compartimenti vicini invii un *ackRKS* di risposta, questo previene uno spreco di risorse computazionali che risulta fondamentale se immaginiamo il caso in cui il middleware MoK sia in esecuzione su un device mobile con risorse limitate.

L'utilizzo di queste primitive implica un'assunzione forte che riguarda il fatto che ogni nodo, compartimento o attore presente nel sistema utilizza una nomenclatura univoca per quanto riguarda i topic, temi, categorie di discorsi ecc..
Questo impone l'utilizzo di un meccanismo che rende le entità consapevoli di ogni cambiamento riguardo i nomi o evita tali cambiamenti per evitare sprechi di risorse computazionali. Abbiamo quindi assunto che tali meccanismi siano incapsulati nel middleware e resi disponibili alle entità per realizzare in maniera trasparente la diffusione.

Descriviamo ora come queste primitive possono essere combinate per garantire l'assunzione di conoscenza ad un attore: La Figura 1.2 mostra come

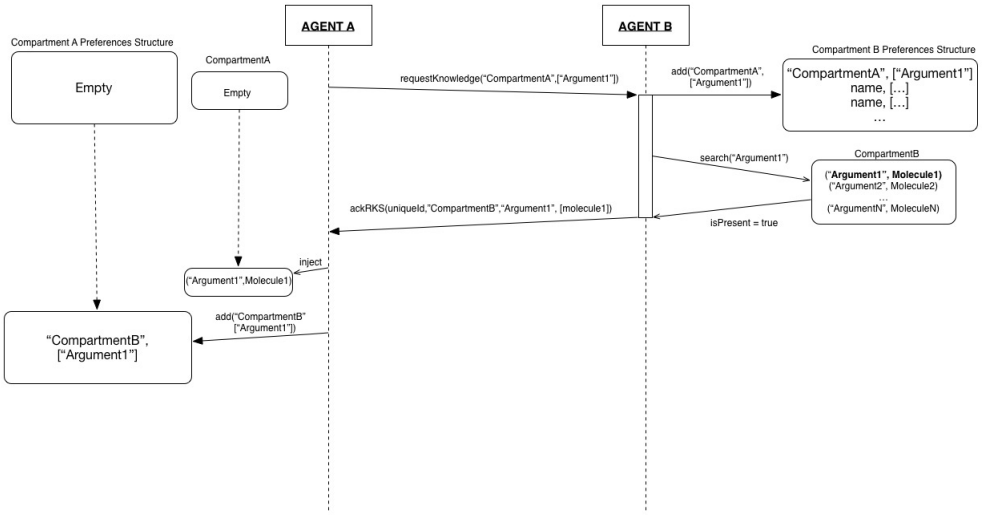


Figura 1.2: Interazione con le primitive

un agente A può recuperare una molecola di conoscenza da un agente B, il quale si trova nel vicinato di A. Di particolare interesse è l'interazione tra gli agenti che avviene in maniera asincrona, infatti A potrebbe eseguire altre azioni mentre B prepara la risposta ed aggiorna le proprie informazioni riguardo ad A. Inoltre sia A che B prima di computare i messaggi ricevuti, potrebbero computarne altri, o terminare le azioni che stavano eseguendo. In altri termini abbiamo assunto la presenza di code di messaggi presenti in entrambe le entità interagenti che fungono da accumulatori, ogni messaggio verrà quindi computato in un secondo momento se necessario. Abbiamo inoltre supposto che il metodo "add", atto all'aggiornamento delle strutture dati, aggiunga elementi in coda senza mai eliminare o sovrascrivere le informazioni già presenti. Nella Figura 1.3 viene mostrato cosa accade se il compartimento di B non detiene alcuna molecola associata all'argomento richiesto da A:

1.1 Problema: Copia o Spostamento di Molecole?

Tale problema consiste nella scelta tra uno di questi due metodi per la condivisione della conoscenza da parte di un compartimento. Fattore che influenza

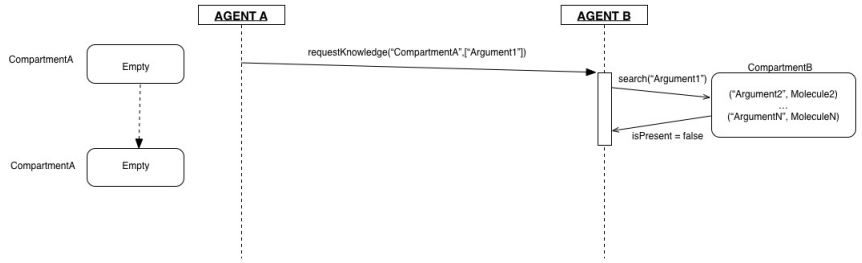


Figura 1.3: Non ci sono molecole per un argomento

tale scelta è sicuramente l'ottimizzazione dell'utilizzo di risorse, in questo caso la memoria, bisogna però tener conto di eventuali problemi di inconsistenza:

- **Copia** : Introduce una forma di ridondanza di informazioni condivise;
- **Spostamento** : Introduce un problema legato alla coordinazione tra compartimenti, se una molecola si sposta tra un compartimento C1 ad un altro C2, nel momento in cui un terzo compartimento C3 richiede quella stessa molecola (che si sta spostando) nessuno risponde alla *requestKnowledge*. Questo problema è stato riportato per completezza ma vogliano indicare che, con le considerazioni che andremo a fare sotto (introducendo la concentrazione) tale problema non sussiste.

Appoggiandoci al modello MoK, possiamo utilizzare una soluzione ibrida tra spostamento e copia per realizzare la diffusione, questo concetto viene realizzato introducendo la concentrazione chimica delle molecole.
Prendendo ispirazione dalla chimica dove la concentrazione delle molecole indica la misura del numero di molecole in una soluzione, la concentrazione in MoK indica il numero di molecole all'interno di un compartimento (soluzione).

Assumiamo che due compartimenti C1 e C2 detengano entrambi la stessa molecola M, dove M(C1)[1] ha concentrazione n ed M(C2)[2] ha concentrazione m.
Ciò introduce quattro possibili tipi di diffusione:

[1]Si intende la molecola M contenuta nel compartimento C1.
[2]Si intende la molecola M contenuta nel compartimento C2.

- **Assorbimento Logico** : Se $n > 1$ ed $m > 0$ allora al termine dell'operazione M(C1) avrà concentrazione n-1 ed M(C2) avrà concentrazione m+1, il compartimento C2 avrà quindi assorbito parte della concentrazione di M da C1.

- **Assorbimento Fisico** : Se $n = 1$ ed $m > 0$ allora al termine dell'operazione M(C1) avrà concentrazione $n = 0$ e la molecola scomparirà da C1 ed M(C2) avrà concentrazione m+1, il compartimento C2 avrà quindi assorbito tutta la concentrazione di M da C1.

- **Cedimento Logico** : Se $n > 1$ ed $m = 0$ allora al termine dell'operazione M(C1) avrà concentrazione n-1 ed M(C2) avrà concentrazione $m = 1$, che significa l'introduzione di M in C2.

- **Cedimento Fisico** : Se $n = 1$ ed $m = 0$ allora al termine dell'operazione M(C1) avrà concentrazione $n = 0$ ed M(C2) avrà concentrazione $m = 1$, che significa l'introduzione di M in C2 e la scomparsa di essa da C1.

1.2 Diffondere gli Aggiornamenti

Per *Diffondere gli Aggiornamenti* si intende l'azione con il quale un compartimento diffonde la notifica di aggiornamento. Quando un compartimento riceverà tale notifica, egli dovrà eseguire gli aggiornamenti solamente se è interesato all'argomento di una molecola, altrimenti dovrà diffondere l'aggiornamento al suo vicinato.

Facendo riferimento a tali concetti la primitiva *diffuseUpdates* sarà definita come segue:

Primitive 3.1.

diffuseUpdates(uniqueId,Molecule,Concentration,Mode)

- **uniqueId** : Rappresenta l'identificatore univoco di una transazione di aggiornamento. Utilizzandolo, possiamo risolvere il problema degli aggiornamenti duplicati nel caso in cui si hanno due o più compartimenti che ne "aggiornano" un altro connesso a loro;

- **Molecule** : Rappresenta la molecola aggiornata.

- **Concentration** : Rappresenta la concentrazione della molecola aggiornata.
- **Mode** : La diffusione della molecola avviene in maniera differente a seconda della modalità specificata (random, vicinato, mirata, probabilistica).

La semantica di questa primitiva implementa i concetti esposti nel paragrafo precedente che verranno utilizzati da un livello più basso a cui il middleware MoK si appoggia.

Attraverso tali primitive possiamo risolvere i problemi riguardanti al *SingleHop*, ad esempio quando un compartimento non è interessato all'argomento proposto da una primitiva, questo viene scartato, il che implica una visione limitata (al vicinato) rispetto a quella reale quando si tratta una rete con migliaia di nodi.

1.3 SingleHop e MultiHop

L'analisi della diffusione nel middleware MoK ci porta a definire due aspetti che sono fondamentali per la diffusione delle informazioni.

Per *SingleHop* all'interno di un contesto di rete o middleware, si intende una trasmissione dei dati attraverso due nodi vicini connessi dal sistema senza *forwarding*. Invece per *MultiHop* si intende una trasmissione dei dati attraverso due nodi connessi ad un massimo numero fissato di nodi intermedi dipendente dal numero di *hop set*.

Se il middleware si basasse sul SingleHop, si formerebbe un sistema dove la diffusione può avvenire solo con i vicini (fisici o logici) e non ulteriormente. Invece con il MultiHop, la diffusione si propaga dalla sorgente ai vicini che poi propagano ai vicini ancora, etc.. Solo che la gestione errata del MultiHop può portare a un intasamento della rete quando un vicino A invia a un vicino B che ha solo come vicini A e C. Infatti se gestito male, B potrebbe inviare comunque a entrambi i vicini il messaggio nonostante A sia colui che l'ha inviato.
Per evitare ciò è assolutamente necessario avere un identificatore che ci permetta di riconoscere se il nostro messaggio è già stato ricevuto.
Un'altra pratica che è possibile utilizzare è quella del gradiente. Si definisce gradiente un quantificatore che identifica il numero di nodi che un messaggio ha attraversato prima di arrivare a quel nodo.

Presa ad esempio la rete in figura, i numeri rappresentano i gradienti. Nel caso di SingleHop, il gradiente massimo sarebbe 1, ossia tutti quei nodi che

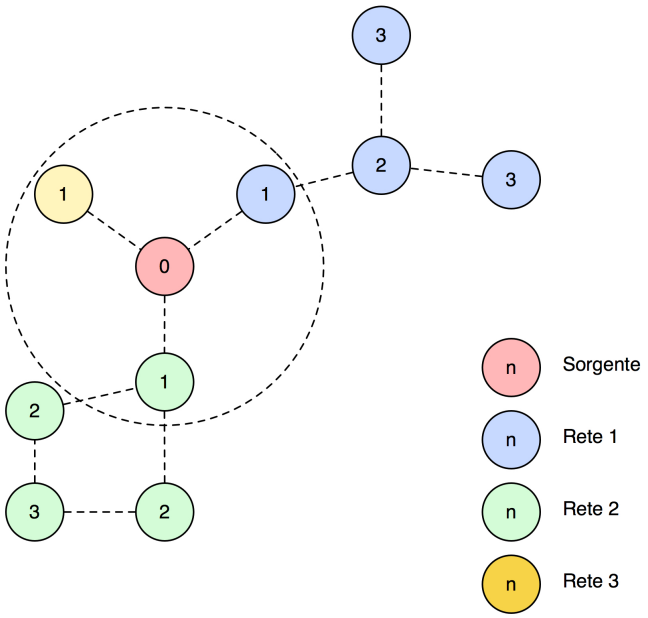


Figura 1.4:

sono connessi direttamente al nodo sorgente.
Invece i nodi con gradiente maggiore di uno, sono quei nodi che non sono collegati direttamente alla sorgente, ma che con *hop* tra i nodi riescono quindi a creare un collegamento indiretto.

Definito questo concetto, è chiaro che il gradiente può essere utilizzato per limitare la diffusione all'infinito attraverso una rete con una moltitudine di punti e gradiente molto alto.

1.3.1 Estensioni primitive per MultiHop

Le primitive definite in precedenza quindi devono essere estese in modo tale da coprire anche il MultiHop in quanto così definite non riescono a ricoprire le problematiche affrontate in precedenza.

Si parte quindi dalla prima primitiva vista cioè *requestKnowledge* e si aggiunge oltre al gradiente anche un identificatore unico condiviso per tutta la rete:

Primitive 1.2.

reqK[3]**(*parameter*[4],compartmentName,[interestedArgs],*grad*,*nHop*)**

Il parametro *grad*[5] sarà impostato dalla sorgente il quale indicherà il numero massimo di *hop* che deve effettuare. In ogni nodo che attraversa, quando viene inoltrato, *grad* deve essere decrementato di 1. Quando si avrà *grad* pari a 0, allora il pacchetto non deve essere ulteriormente inoltrato.

Il parametro *nHop* invece deve funzionare in maniera opposta, ossia parte sempre da zero e ogni volta che la richiesta viene inoltrata si incrementa di 1. Questo parametro ci serve per la risposta in modo da avere il numero di salti effettuati fino a quel punto.

La seconda primitiva da essere modificata è:

Primitive 2.2.

ackRKS(*parameter*[6],compartmentName,Argument,[Molecule], *grad*)

Il parametro *grad* invece sarà impostato inizialmente al valore del parametro *n* di *requestKnowledge* e funziona esattamente come il parametro *grad* di tale primitiva, ossia viene decrementato ogni volta che la risposta viene inoltrata ed è accettato solo quando vale 0. In questo modo siamo sicuri che ripercorrendo gli *hop* in senso contrario, la diffusione della risposta tornerà indietro a chi ha inviato la richiesta.

Lista di routing

Come notato nella sezione precedente, fra i primi parametri delle funzioni vi è scritto *parameter*. Questo indica che ci sarà una scelta legata al valore che si vuole realmente come parametro della funzione. Nel caso SingleHop è stato semplice pensare la mancanza di necessità di parametri in quanto si conoscevas il mittente della richiesta, perchè si trattava solamente con i nodi

[3]Si intende *requestKnowledge*

[4]Spiegata nella sottosottosezione successiva

[5]`http://www.agentgroup.unimo.it/didattica/cas/L13/Mamei_TOTA.pdf`, Mamei M., Zambonelli F.

[6]Spiegata nella sottosottosezione successiva

vicini. Ora invece con il MultiHop affrontiamo un problema che riguarda soprattutto come inviare la risposta attraverso una serie di collegamenti non conosciuti a priori. Il metodo più semplice per evitare un invio casuale e broadcast di molecole può essere l'uso della **lista di routing**.

La lista di routing è una struttura dati formata dai nodi che a partire dalla sorgente, ogni volta che una richiesta viene inoltrata da un nodo, aggiungono il loro identificatore alla lista che poi andranno a passare come parametro della richiesta inoltrata. Così facendo quando la richiesta arriverà al nodo che effettivamente ha le informazioni utili nel proprio compartimento, la risposta potrà seguire la "scia" di routing percorsa dalla richiesta e quindi tornare al nodo che l'ha inviata.

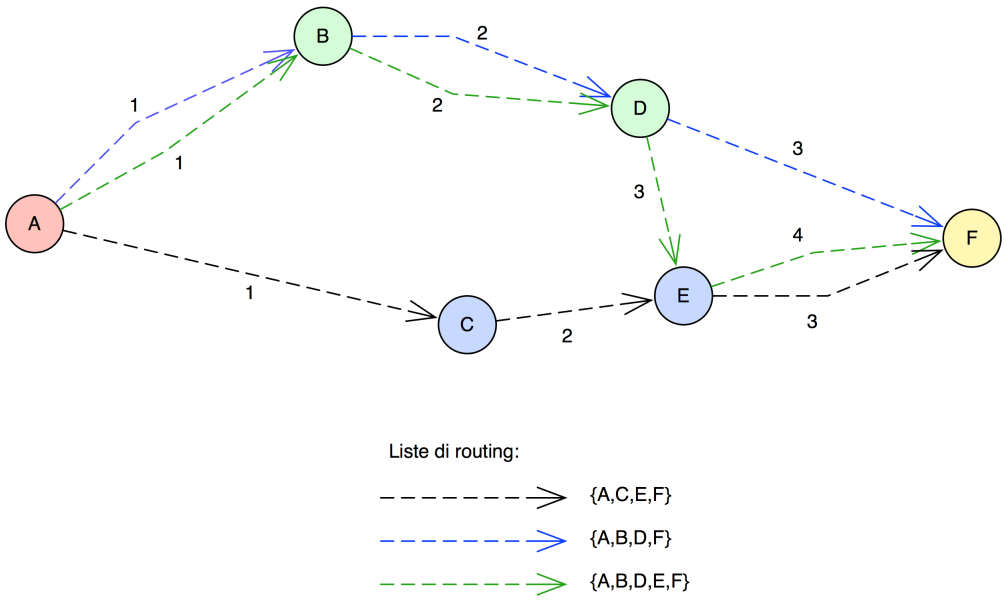


Figura 1.5: Esempio di creazione della lista di routing

Nella Figura 1.5 si vede come una richiesta da parte di A con la lista di routing, arriva in F in diverse forme a seconda del percorso. Quindi se F o uno dei nodi della rete ha qualche molecola riferita all'argomento, sa come dovrà rispondere e che percorso dovrà effettuare per rispondere alla richiesta della sorgente, quindi con l'utilizzo di tale strategia la primitiva sarà simile a quella del SingleHop e non si avrà il problema della duplicazione della risposta alle richieste. Nella Figura 1.5, F ha diverse vie che può percorrere per rispondere ad A, ma una volta che arriva la prima risposta, A sta aspettando eventuali altre risposte per quella richiesta ma non da F, in

quanto egli ha memorizzato l'ultimo elemento lista di routing arrivata nella risposta più veloce in una struttura dati e sa che dal nodo F ha già ricevuto una risposta per quell'argomento. In tal modo eventuali risposte ridondanti verranno scartate. Quando successivamente si effettua una nuova richiesta per un argomento di natura diversa, allora bisognerà pulire la struttura dati contenente l'insieme di nodi che hanno risposto all'argomento precedente in quanto non più necessari.

Primitive 1.3.

reqK(*listRouting*,compartmentName,[interestedArgs],*grad*,*nHop*)

Primitive 2.3.

ackRKS(*listRouting*,compartmentName,Argument,[Molecule], *grad*)

grad in *ackRKS* diminuisce come spiegato in precedenza, e questo ci fornisce la posizione nella lista di routing del nodo stesso. Ad esempio, nella Figura 1.4 il nodo C quando riceve l'*ackRKS* controlla se in posizione 1 della lista di routing di colore blu egli è presente. Se questo è verificato allora inoltra, altrimenti scarta.

diffuseUpdate

Questa primitiva, a differenza delle altre due, non ha bisogno della lista di routing in quanto è una primitiva che vuole solo modificare una molecola e non ha bisogno di ricevere risposta. Il *MultiHop* però comporta anche la possibilità di impostare il gradiente dell'operazione di aggiornamento che va quindi aggiunto tra i parametri. Pertanto l'ultima primitiva da modificare è *diffuseUpdates*.

Primitive 3.2.

diffuseUpdates(uniqueID,Molecule,Concentration,Mode,*grad*)

Questa operazione userà un *uniqueID* condiviso in modo che se un aggiornamento arriva due volte allo stesso argomento, quest'ultimo viene scartato[7]. Il parametro *grad* verrà usato allo stesso modo delle primitive precedenti.

[7]Problema del Flooding

1.4 Traces

Le tracce [8] sono state introdotte per simulare gli effetti collaterali dovuti all'esecuzione di azioni. Queste si differenziano in base al tipo a seconda del loro enzima "padre" e sono osservabili solo dalle reazioni MoK (e non dai catalizzatori), sono inoltre soggette a diffusione, al decadimento e alle reazioni di perturbazione "enzyme-dependent".

Le tracce sono libere di girare nella rete dei compartimenti MoK cercando un'occasione per applicare la loro reazione di perturbazione, abilitano quindi l'ambiente non solo a tener traccia delle azioni degli utenti, ma anche a sfruttarle approfittando del processo di coordinazione. Quindi, le tracce partecipano alla diffusione in MoK in maniera proattiva, ovvero l'azione avviene in maniera autonoma senza una richiesta esplicita, questo è già stato modellato nel caso della diffusione di molecole pertanto utilizzeremo gli stessi concetti per la diffusione delle tracce.

Siccome lo scopo di questo tipo di astrazione è diverso da quello precedentemente studiato per la diffusione di molecole, si necessita l'aggiunta di un caso d'uso apposito per modellare la reazione di diffusione per le tracce.

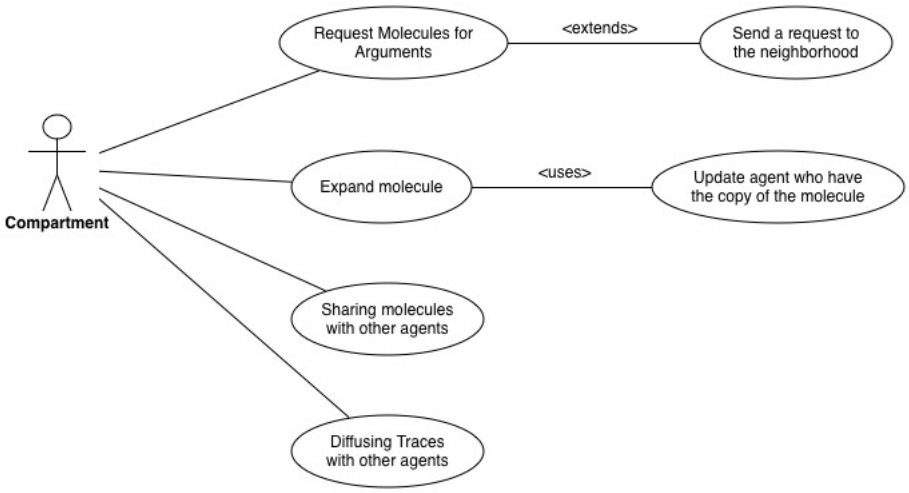


Figura 1.6: Case study with traces

[8] Vedi *Anticipatory Coordination in Socio-technical Knowledge-intensive Environments: Behavioural Implicit Communication in MoK*

Dati i seguenti casi d'uso abbiamo deciso di accorpare *Request Molecules for Arguments* e *Expand molecule* con le relative estensioni per dare una rappresentazione più chiara e compatta dei possibili scenari in cui un compartimento può trovarsi.

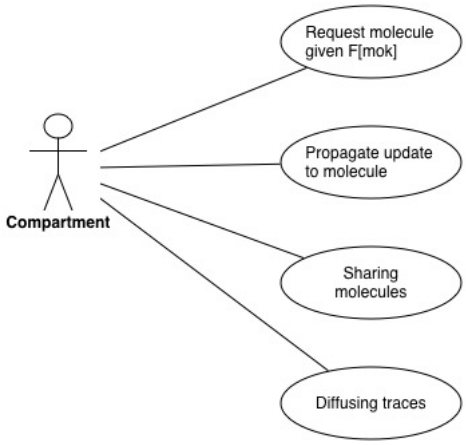


Figura 1.7: Case study Updated

Inoltre introduciamo l'utilizzo di *F[MoK]*[9] che esprime la similarità fra molecole e ci permette di astrarre dall'argomento/topic della molecola, pertanto *reqK* assumerà la seguente forma:

Primitive 1.4.

reqK(*listRouting*,compartmentName,F[MoK],Concentration,*grad*,*nHop*)

- **listRouting** : vedi sopra;
- **compartmentName**: vedi sopra;
- **F[MoK]**: funzione di similarità semantica tra molecole
- **Concentration** : vedi sopra.
- **grad** : vedi sopra.[10].
- **nHop** : vedi sopra.

[9]Vedi la presentazione *Molecules of Knowledge Self-Organisation in Knowledge-Intensive Environments*

[10]Vedi Multi-Hop

Primitive 2.4.

respK(*listRouting*,compartmentName,F[MoK],[Molecule], *grad*)

Come denotato dalle primitive, gli *Arguments/topic* sono stati sostituiti con *F[MoK]*, inoltre la primitiva *respK* evidenziata qui sopra fa riferimento a *ackRKS* che abbiamo introdotto precedentemente. La differenza fra le due primitive verrà esplicitata in seguito.
Analizzando le tracce è emerso che, nonostante lo scopo sia diverso da quello delle molecole, il metodo di diffusione è simile a quello analizzato in precedenza per la diffusione degli aggiornamenti delle molecole. Pertanto sarà necessario creare una nuova primitiva in quanto, anche se simili, in aggiornamenti futuri del middleware si potrebbero modellare in maniera differente l'implementazione di tali primitive.

Primitive 4.1.

diffuseTraces(*listRouting*,Trace,compartmentName,F[MoK], Concentration,*grad*)

- **listRouting** : vedi *MultiHop*;
- **Trace** : Rappresenta la traccia rilasciata nel compartimento in cui è avvenuta l'azione;
- **compartmentName**: è il nome del compartimento che diffonde la traccia.
- **F[MoK]**: funzione di similarità semantica tra molecole
- **Concentration** : Rappresenta la concentrazione della traccia.
- **grad** : valore del gradiente[11].

Tale primitiva si comporta esattamente come la *diffuseUpdates* esplicitata precedentemente ma rispetto ad essa non vi è la necessità di adottare un identificatore univoco in quanto si assume che è il middleware sottostante ad occuparsi di evitare il flooding.
Essendo MoK un *socio-technical Knowledge-Intensive Environment*, deve poter fornire agli utenti i mezzi per interagire con il sistema stesso usando i BIC

[11] Vedi Multi-Hop

tacit message[12]. Con l'introduzione delle tracce, possiamo utilizzare la reazione di perturbazione per inviare messaggi con lo scopo di scoprire la topologia della rete, per stabilire canali di comunicazioni privilegiati, per abilitare o disabilitare alcuni protocolli di interazione o per notificare l'utente sulla disponibilità di nuove informazioni. Perciò il middleware MoK deve garantire per la diffusione una primitiva che aiuti i vari compartimenti a diffondere questo tipo di messaggi. Riguardando quanto fatto in precedenza, si è notato che la primitiva di *ackRKS* ha un comportamento simile a quello che vorremo per questa estensione e si è deciso di farle assumere una semantica aggiuntiva a quella introdotta precedentemente, ovvero quella descritta dal seguente diagramma di interazione:

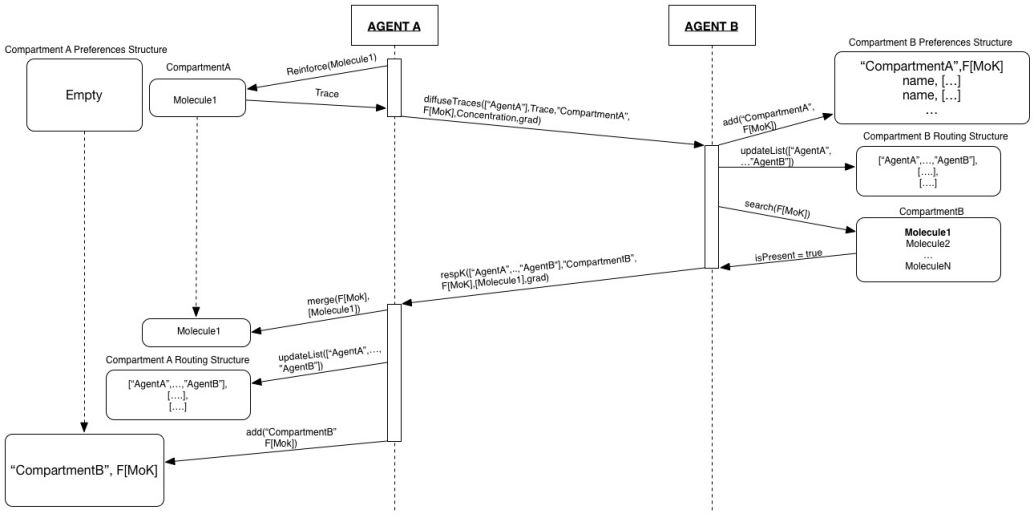


Figura 1.8: Scenario di interazione per la diffusione di tracce ed eventuale risposta.

Introduciamo ora lo scenario duale a quello presentato qui sopra, cioè quello in cui un compartimento esegue un'azione su una molecola e il comportamento tramite la *diffuseTraces* diffonde il messaggio di notifica dell'azione avvenuta. A questo punto il compartimento che riceve tale messaggio decide, dato che anche lui è interessato alla molecola, di richiederla direttamente al compartimento che ha diffuso la traccia. Ma come modellata in precedenza, la richiesta di molecole non è mai mirata, dato che la topologia della rete è ignota, ma è una richiesta broadcast. Per tale motivo si è deciso di modellare nuovamente la primitiva di richiesta aggiungendo un nuovo parametro:

[12]Vedi *Anticipatory Coordination in Socio-technical Knowledge-Intensive Environments: Behavioural Implicit Communication in MoK*

Primitive 1.5.

reqK(listRouting,compartmentName,F[MoK],Concentration, grad,nHop,*mode*)

dove ***mode*** è la modalità con cui la richiesta viene inviata. Può essere di due tipi:

- *REQ_BROADCAST* : uguale alla *reqK* vista nelle modellazioni precedenti;
- *REQ_FOCUSED*: la richiesta segue la lista di routing per effettuare una richiesta mirata a un compartimento specifico.

Una volta specificato questo aggiornamento della *reqK*, il diagramma delle interazioni dello scenario descritto in precedenza sarà come mostrato in Figura 1.9.

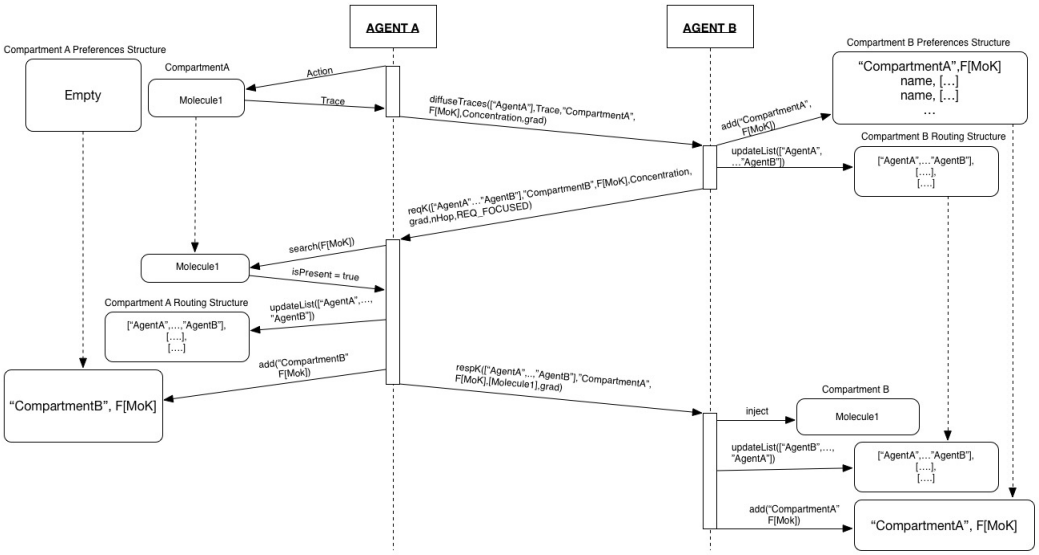


Figura 1.9: Caso duale a quello presentato nella Figura 1.8.

Questi scenari mostrano come la modifica alle primitive ci permetta di colmare le possibilità di utilizzo del sistema dopo aver introdotto le tracce, ovvero grazie ad esse un compartimento può far conoscere le proprie esigenze/preferenze ad altri che eventualmente (a seconda del *tacit message*) possono rispondere con informazioni aggiuntive. Di particolare interesse è il fatto che i due nodi A e B possono anche non "vedersi" all'interno del proprio

vicinato (Multi-Hop), questo grazie all'introduzione di apposite strutture come la lista di routing e il gradiente. Tali informazioni risultano utili anche per ulteriori scopi. Se ad esempio la comunicazione fra A e B per qualche ragione è molto frequente, essi avrebbero necessità di "scoprire" più volte il cammino che li collega. Questo porta ad inevitabili rallentamenti nella rete e pertanto necessitiamo l'introduzione di un'apposita struttura dati *Compartment X Routing Structure* che promuove la memorizzazione di un numero limitato di cammini per evitare di doverli ricercare più volte in caso siano molto frequenti. A fronte dell'invio di una possibile risposta quindi, prima di tutto il middleware controlla se un cammino è già stato percorso ed in tal caso procede senza doverlo ricostruire "nodo per nodo" utilizzando le tecniche viste in precedenza.
Dato che però la rete è dinamica e la topologia potrebbe variare nel tempo è necessario introdurre uno studio per determinare il livello di aggiornamento delle informazioni presenti in tale struttura, per non ricadere nel caso di cammini non raggiungibili ad esempio.

1.4.1 Metodi per mantenere aggiornati i cammini

Come ipotizzato sopra, utilizziamo un'apposita struttura dati per gestire le comunicazioni molto frequenti fra due nodi della rete. Tale struttura conterrà pertanto l'insieme di nodi che formano il cammino fra quello di partenza e quello di arrivo, essi compresi. Questo ci ha portato a rappresentare un cammino nel seguente modo:

$$("Node1", "Node2", ..., "NodeN") \tag{1.2}$$

Dove *Node1* è il nodo di partenza e *NodeN* è il nodo di arrivo.
Un compartimento detiene una propria struttura dati atta alla memorizzazione di un numero limitato di cammini, scelto in base alle risorse disponibili. Forniamo ora i dettagli relativi alla gestione dei cammini presenti nella struttura dati e di come questa verrà poi gestita all'interno del Middleware MoK.

- **Aggiornamento temporale :**
  Prevede l'utilizzo periodico del cammino presente nella struttura dati, ovvero se all'istante T0 all'interno della struttura appartenente al compartimento C1 viene aggiornato il cammino per raggiungere un compartimento C2 e all'istante T1, C1 cerca di "contattare" C2 nuovamente, allora se T1-T0 <= T, dove T è una costante predefinita, allora si può procedere utilizzando le informazioni presenti, altrimenti si dovrà utilizzare un nuovo cammino che deve essere scoperto.

A tale scopo abbiamo la necessità di tener traccia, non solo del cammino, ma anche dell'istante di tempo in cui questo viene inserito o aggiornato nella struttura dati.
Pertanto se si decide di adottare questa strategia di aggiornamento, le istanze presenti nella struttura dati da associare al compartimento MoK assumeranno tale forma:

$$[(\text{"}Node1\text{"}, \text{"}Node2\text{"}, ..., \text{"}NodeN\text{"}), T0] \tag{1.3}$$

Dove per far fronte alle necessità sopra esplicitate un cammino viene rappresentato con la tupla composta dalla lista di nodi e un *Timestamp*, in tal modo a fronte di una nuova richiesta di comunicazione verso *"NodeN"* la semantica appena introdotta prevede la ricerca tra tutti gli elementi contenuti nella struttura dati del cammino che ha come nodo finale *"NodeN"*.
Se esiste tale elemento, si confronta la differenza fra il timestamp attuale e *T0* con la costante T descritta precedentemente e si decide se utilizzare questo cammino come lista di Routing da passare come parametro alla primitiva di richiesta o eventualmente ricalcolarlo passando una lista vuota.

- **Aggiornamento per disconnessione :**
  Prevede la necessità di aggiornare le informazioni presenti nella struttura solo dopo l'avvento della disconnessione del nodo su cui si effettua il primo "salto" del cammino corrispondente come mostrato in Figura 1.10.

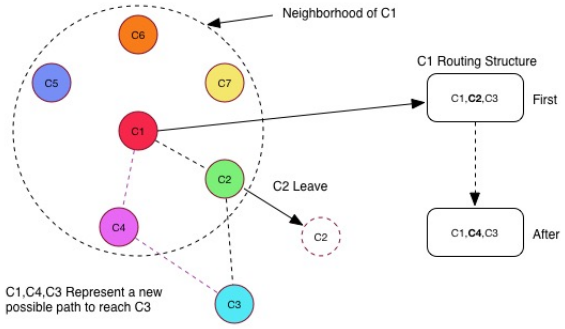


Figura 1.10: C2 esce dal vicinato di C1 che necessita un nuovo cammino per raggiungere C3.

Se consideriamo la struttura dati di Routing di C1 prima dello spostamento di C2 fuori dal vicinato, allora C2 rappresenta il nodo su cui C1 deve effettuare il "primo salto" per raggiungere C3. Una volta uscito C2, C1 dovrà ricalcolare il percorso per raggiungere C3 se necessita di comunicare nuovamente con esso.

- **Aggiornamento per nuove connessioni :**
  L'aggiornamento dei cammini deve avvenire dopo un certo numero di nuove connessioni N riscontrate nel proprio vicinato, ovvero se consideriamo N=2 e il compartimento C1, allora la struttura dati associata a C1 verrà aggiornata solo quando altri 2 compartimenti che non si trovano nel suo vicinato, ne entrano a far parte. Questo permette di mantenere coerenza con la topologia dinamica della rete.

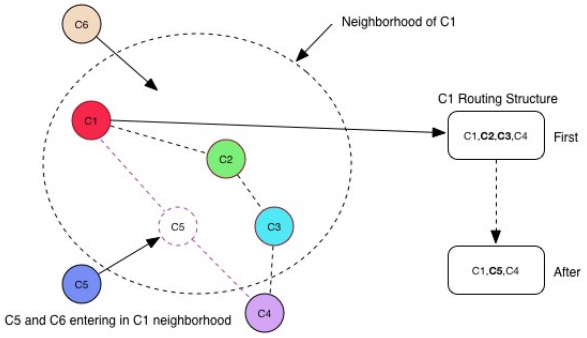


Figura 1.11: N=2, C5 e C6 entrano nel vicinato di C1 che è costretto, ma in questo caso è corretto, a ricalcolare il cammino.

Nel caso di *Aggiornamento per nuove connessioni* e *Aggiornamento per disconnessione* non si necessita di alcuna informazione aggiuntiva per effettuare l'aggiornamento della struttura dati (oltre al cammino) la cui forma rimane pertanto invariata. In entrambi i casi è però necessario che il compartimento tenga traccia del numero di connessioni/disconnessioni per effettuare gli opportuni aggiornamenti quando necessario, i quali prevedono l'eliminazione di tutti i cammini dalla struttura dati.
Pertanto si prevede l'utilizzo di una variabile, da inserire nel compartimento MoK, che si aggiorna a fronte dell'accadimento di un evento[13] opportuno e viene sfruttata per controllare il superamento di una soglia prefissata il cui

[13]Si intende l'evento di connessione o disconnessione, la cui gestione è demandata al middleware MoK.

effetto prevede il ricalcolo di tutti i percorsi che nel frattempo potrebbero essere cambiati.

1.5 Riepilogo

Vista la modellazione fatta finora, riassumiamo nella forma più aggiornata le primitive analizzate come illustrato nella Figura 1.12.

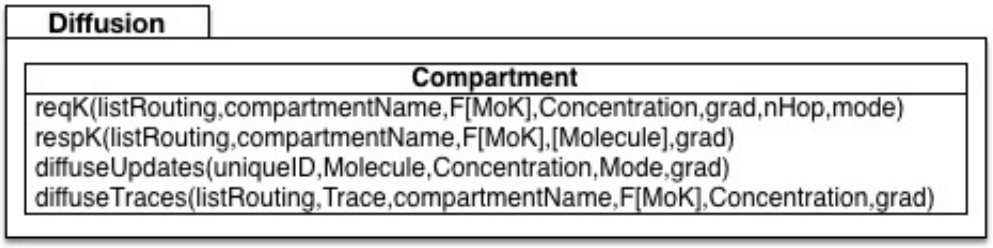


Figura 1.12: Riepilogo primitive

Capitolo 2

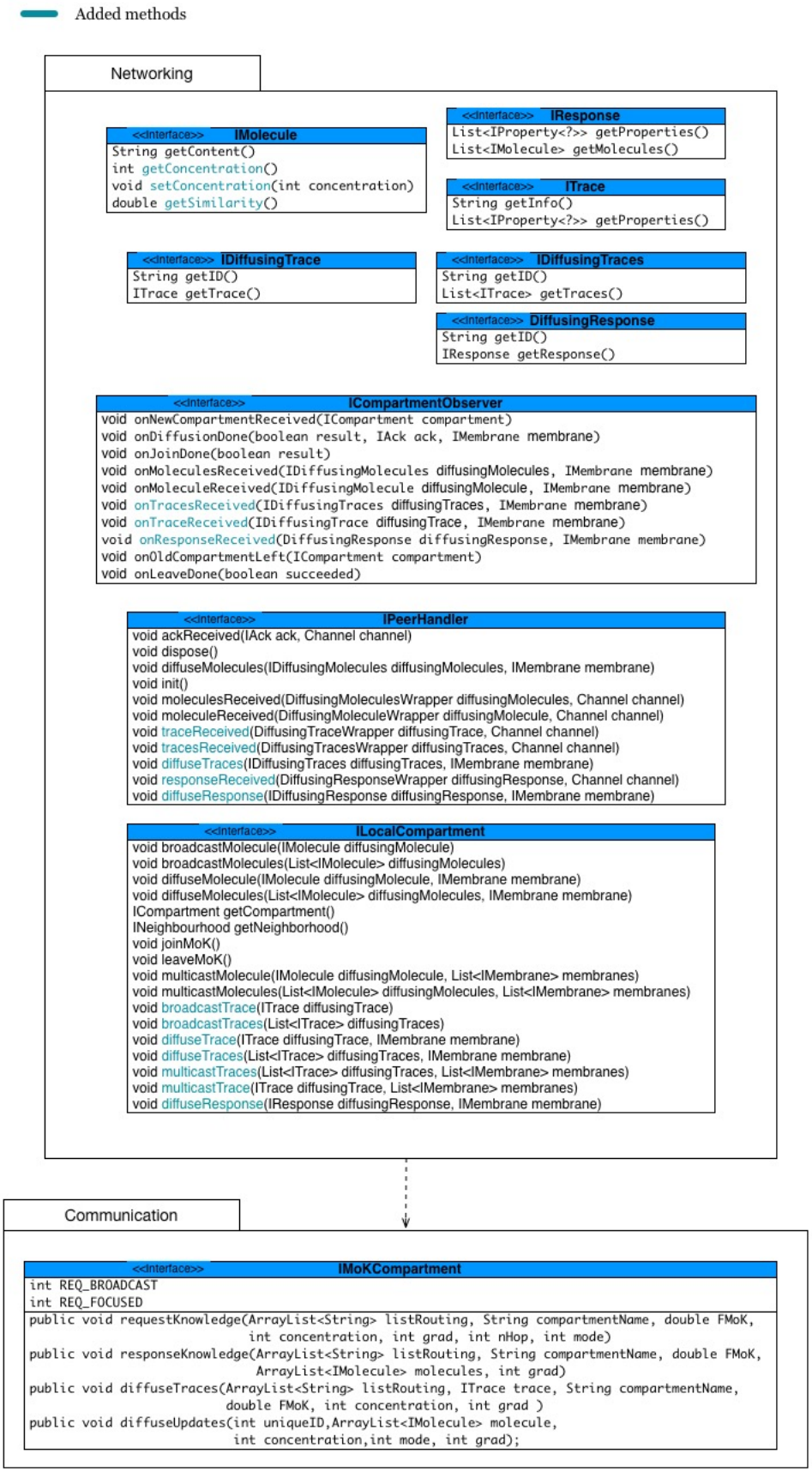
Fase di Progetto

Per implementare le primitive risultanti dalla precedente fase di analisi ci siamo appoggiati all'implementazione fornitaci dal branch di MoK Networking[1]. In particolare abbiamo inanzitutto analizzato le funzionalità fornite nell'implementazione attuale e ci siamo accorti che non tutto il necessario era già stato implementato, introduciamo quindi una prima fase di refactoring.

2.1 Refactoring del progetto Networking

Come fornita inizialmente l'implementazione del progetto Networking prevedeva solamente la comunicazione di molecole fra ad esempio alcune entità Client chiamate *Peer* connesse fra loro, per incapsulare il concetto del vicinato, attraverso delle entità *HotSpot* che fungono da Server la cui locazione è ben conosciuta dai Client. Dalla nostra analisi è risultato che le molecole non sono le uniche entità passive a poter viaggiare fra i *Peer*, ma abbiamo bisogno anche di poter scambiare informazioni fra i compartimenti attraverso l'utilizzo di Tracce. A tale scopo abbiamo quindi introdotto il concetto di traccia attraverso l'apposita interfaccia e gestito la comunicazione di conseguenza.

[1] Vedi *RelazioneDradi1.3*


Added methods
Networking
<<Interface>> IMolecule
String getContent()
int getConcentration()
void setConcentration(int concentration)
double getSimilarity()
<<Interface>> IResponse
List<IProperty<?>> getProperties()
List<IMolecule> getMolecules()
<<Interface>> ITrace
String getInfo()
List<IProperty<?>> getProperties()
<<Interface>> IDiffusingTrace
String getID()
ITrace getTrace()
<<Interface>> IDiffusingTraces
String getID()
List<ITrace> getTraces()
<<Interface>> DiffusingResponse
String getID()
IResponse getResponse()
<<Interface>> ICompartmentObserver
void onNewCompartmentReceived(ICompartment compartment)
void onDiffusionDone(boolean result, IAck ack, IMembrane membrane)
void onJoinDone(boolean result)
void onMoleculesReceived(IDiffusingMolecules diffusingMolecules, IMembrane membrane)
void onMoleculeReceived(IDiffusingMolecule diffusingMolecule, IMembrane membrane)
void onTracesReceived(IDiffusingTraces diffusingTraces, IMembrane membrane)
void onTraceReceived(IDiffusingTrace diffusingTrace, IMembrane membrane)
void onResponseReceived(DiffusingResponse diffusingResponse, IMembrane membrane)
void onOldCompartmentLeft(ICompartment compartment)
void onLeaveDone(boolean succeeded)
<<Interface>> IPeerHandler
void ackReceived(IAck ack, Channel channel)
void dispose()
void diffuseMolecules(IDiffusingMolecules diffusingMolecules, IMembrane membrane)
void init()
void moleculesReceived(DiffusingMoleculesWrapper diffusingMolecules, Channel channel)
void moleculeReceived(DiffusingMoleculeWrapper diffusingMolecule, Channel channel)
void traceReceived(DiffusingTraceWrapper diffusingTrace, Channel channel)
void tracesReceived(DiffusingTracesWrapper diffusingTraces, Channel channel)
void diffuseTraces(IDiffusingTraces diffusingTraces, IMembrane membrane)
void responseReceived(DiffusingResponseWrapper diffusingResponse, Channel channel)
void diffuseResponse(IDiffusingResponse diffusingResponse, IMembrane membrane)
<<Interface>> ILocalCompartment
void broadcastMolecule(IMolecule diffusingMolecule)
void broadcastMolecules(List<IMolecule> diffusingMolecules)
void diffuseMolecule(IMolecule diffusingMolecule, IMembrane membrane)
void diffuseMolecules(List<IMolecule> diffusingMolecules, IMembrane membrane)
ICompartment getCompartment()
INeighbourhood getNeighborhood()
void joinMoK()
void leaveMoK()
void multicastMolecule(IMolecule diffusingMolecule, List<IMembrane> membranes)
void multicastMolecules(List<IMolecule> diffusingMolecules, List<IMembrane> membranes)
void broadcastTrace(ITrace diffusingTrace)
void broadcastTraces(List<ITrace> diffusingTraces)
void diffuseTrace(ITrace diffusingTrace, IMembrane membrane)
void diffuseTraces(List<ITrace> diffusingTraces, IMembrane membrane)
void multicastTraces(List<ITrace> diffusingTraces, List<IMembrane> membranes)
void multicastTrace(ITrace diffusingTrace, List<IMembrane> membranes)
void diffuseResponse(IResponse diffusingResponse, IMembrane membrane)
Communication
<<Interface>> IMoKCompartment
int REQ_BROADCAST
int REQ_FOCUSED
public void requestKnowledge(ArrayList<String> listRouting, String compartmentName, double FMoK,
int concentration, int grad, int nHop, int mode)
public void responseKnowledge(ArrayList<String> listRouting, String compartmentName, double FMoK,
ArrayList<IMolecule> molecules, int grad)
public void diffuseTraces(ArrayList<String> listRouting, ITrace trace, String compartmentName,
double FMoK, int concentration, int grad)
public void diffuseUpdates(int uniqueID,ArrayList<IMolecule> molecule,
int concentration,int mode, int grad);

Come mostrato in figura il nostro approccio ha previsto la duplicazione dei metodi relativi alla gestione della comunicazione fra molecole per poter gestire anche le tracce, inoltre per poter verificare e modificare le proprietà di una molecola sono stati aggiunti gli appositi metodi nell'interfaccia *IMolecule*. Per poter evitare di gestire l'introduzione ad altri problemi, dovuti all'invio ad esempio di una traccia ed una molecola, durante l'implementazione abbiamo deciso di introdurre il tipo di dati *IResponse* che funge da risposta alla richiesta di conoscenza, in particolare abbiamo incapsulato all'interno di questa entità le molecole da inviare al compartimento richiedente e una lista di proprietà in cui saranno presenti le informazioni relative al forwarding della richiesta.

Per implementare l'entità *MoKCompartment* abbiamo utilizzato il pattern composition, quindi senza effettivamente estendere l'entità *LocalCompartment* fornitaci da Networking, abbiamo incapsulato un'istanza di tale entità all'interno del nostro compartimento. Dato che *MoKCompartment* risulta in ogni caso un'astrazione del compartimento questo fornirà le stesse funzionalità proposte dal *LocalCompartment* in aggiunta e supporto alle nostre primitive.

2.1.1 Strutture Dati Utilizzate

Dalla fase di analisi risulta evidente la necessità di utilizzare alcune strutture dati come memoria temporanea(caching) per poter memorizzare alcune informazioni, come la conoscenza preferita dei compartimenti o la posizione/presenza di altri compartimenti nella rete(lista di Routing). Pertanto abbiamo previsto una serie di strutture dati all'interno del *MoKCompartment*:

```
/*
 * need to save the neighborhood preferences
 * (Compartment UUID, list of similarity MoK Function)
 */
private HashMap<String,List<Double>> preferences;
/*
 * Knowledge held by the Compartment
 */
private CopyOnWriteArrayList<IMolecule> knowledge;
/*
 * need to save the Routing information about a compartment and the timestamp when inserted and the hit
 * (Compartment UUID, list of nodes(Routing List))
 */
@SuppressWarnings("rawtypes")
private HashMap<String,ArrayList<IProperty>> RoutingInfoCache;
```

Figura 2.1: Strutture Dati

In cui **preferences** permette, a fronte del fatto che un compartimento presente nella rete ha effettuato una richiesta, di memorizzare tale evento nel compartimento corrente ricordando la similarità presente nella richiesta. Se per un qualche motivo il nostro compartimento vorrà ottenere in futuro informazioni con quella stessa similarità, potrà quindi effettuare una richiesta mirata. Per il momento l'implementazione di questa cache prevede solamente la memorizzazione ed è stata realizzata come utile per sviluppi futuri.
La struttura dati **knowledge** rappresenta la conoscenza del compartimento, ovvero l'insieme di molecole presenti correntemente all'interno di esso, quando si vuole verificare a fronte di una richiesta la presenza o meno di molecole simili la ricerca verrà fatta su questa struttura dati, andando a confrontare la similarità della richiesta con quella delle singole molecole presenti nella struttura.
RoutingInfoCache è utile al fine di memorizzare le liste di Routing che vengono recuperate/inserite ogni volta che il compartimento riceve una richiesta. In particolare abbiamo incapsulato questa informazione all'interno di una lista di proprietà in cui ci saranno altri parametri come il *Timestamp* ed il numero di *Hit* utili al fine di mantenere aggiornata la cache, questi vengono aggiornati ogni qualvolta viene utilizzata una particolare lista di Routing e ci permettono di evitare la persecuzione percorsi obsoleti che, qualora dovesse capitare la mancanza di un qualche nodo all'interno della rete, questo avvenimento causerà l'invalidazione della lista corrente. Questa può essere vista come una possibile via per aggiornare le liste di routing ed eliminare i percorsi vecchi e poco intrapresi. Ma purtroppo in MoK, se due compartimenti non sono direttamente connessi, non si riesce a capire se sono ancora presenti o meno. Per fare ciò, è stato implementato un semplice metodo che quando si intraprende un percorso e uno di questi nodi ha lasciato MoK, quando il compartimento prima di questo non trova più nel suo vicinato il compartimento mancante, ripercorre la lista di routing fino al compartimento mittente indicando ad ogni nodo che appartiene alla lista di routing che il nodo in questione non è più presente. Quindi quando riceve tale messaggio il compartimento procede a cancellare tutte le liste di routing in cui tal nodo è presente. Si vuole far notare che tali strutture dati dovrebbero essere limitare per evitare un'utilizzo improprio della memoria nel caso in cui ad esempio si vuole far eseguire il *Peer* su un device mobile.

2.2 Implementazione di RequestKnowledge

Una volta eseguito con successo il refactoring del progetto Networking abbiamo deciso di implementare la primitiva *RequestKnowledge* attraverso l'u-

tilizzo delle tracce seguendo i seguenti passi:

- Abbiamo previsto un programma di test che presenta la seguente interfaccia grafica:

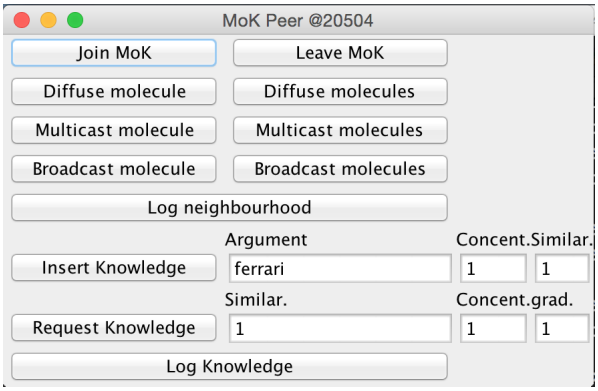


Figura 2.2: TestPeer

  In cui abbiamo dato la possibilità ad un'ipotetico Client di inserire molecole tramite il tasto **Insert Knowledge**, andando a specificare negli appositi campi a fianco le varie proprietà della molecola, in particolare la similarità rappresenta il parametro *FMoK* della primitiva di *requestKnowledge*.

- Dato il punto il precedente abbiamo quindi previsto la possibilità di effettuare una richiesta di conoscenza da parte di un compartimento tramite il tasto **requestKnowledge** che prenderà in ingresso i parametri da specificare negli appositi campi a fianco, in particolare in questa fase abbiamo inteso la semantica della richiesta come segue: se il compartimento conosce la posizione e l'esistenza di altri compartimenti del sistema (forniti tramite una memoria cache prevista anche in fase di analisi atta alla memorizzazione delle varie liste di Routing) allora notifica l'utente e mostra tutte le possibili liste di Routing che ha memorizzato fino ad un dato istante e permette in questo modo di fare una richiesta in modalità *Focused*, se in caso contrario il compartimento non conosce nessun altro, allora verrà eseguita una richiesta in modalità *Broadcast*. In particolare un compartimento viene a conoscenza della posizione/presenza di un altro compartimento del sistema solo quando quest'ultimo effettua una richiesta verso il primo compartimento.

- Una volta scatenata la richiesta questa viaggerà fra i compartimento sotto forma di traccia la quale prevede secondo la nostra modellazio-

ne due proprietà fondamentali: un contenuto informativo che possiamo recuperare con il metodo *getInfo()* specificato in ITrace ed una lista di IProperty che possiamo recuperare con il metodo *getProperties()*. In particolare nel caso di una richiesta la traccia corrispondente assumerà la seguente forma: Info conterrà la stringa "RequestKnowledge" e la lista di proprietà conterrà i parametri della richiesta(listRouting,grad,nHop ecc..) che verranno aggiornati man mano che la richiesta si espande nella rete.

- Quando un compartimento riceve una traccia di "RequestKnowledge" recupera i parametri associati ad essa e di conseguenza aggiorna le strutture dati cache con le opportune informazioni e verifica se è presente la conoscenza richiesta al proprio interno, se questo è il caso esegue una *ResponseKnowledge*. In ogni caso si tenta di effettuare il forwarding della richiesta che effettivamente andrà a buon fine solamente se il gradiente lo permette.

Questi passaggi fondamentali racchiudono la logica implementativa utilizzata nella realizzazione della primitiva *RequestKnowledge*, si vuole evidenziare il fatto che un compartimento che riceve una richiesta ottiene, tramite l'apposita traccia, il parametro *FMoK* che rappresenta la similarità fra molecole. La verifica del fatto che un compartimento contiene una o più molecole richieste è stata totalmente realizzata basandosi unicamente su tale parametro, senza quindi utilizzare anche il Topic per esempio.

2.3 Implementazione di ResponseKnowledge

Per realizzare questa primitiva abbiamo utilizzato una strategia simile a quella specificata per la richiesta con la differenza che grazie alla lista di Routing, passata come parametro in ingresso, il compartimento non dovrà far altro che ripercorrere i nodi presenti in tale lista per poter arrivare al compartimento che effettivamente ha effettuato la richiesta. A tale scopo quindi la semantica implementata è la seguente:

- Quando un compartimento esegue una *ResponseKnowledge* si recupera l'ultimo *CompartmentID* presente nella lista la cui membrana deve essere presente nel compartimento corrente(deve essere un nodo vicino). Una volta recuperata tale membrana ad essa viene inviata una risposta contenente le informazioni per il forwarding e le molecole, incapsulate in un'entità che implementa l'interfaccia *IResponse*.

- A fronte della ricezione di tale pacchetto JSON, il prossimo compartimento nella lista di Routing gestirà la risposta a seconda delle informazioni contenute fra le proprietà del messaggio, o ne viene fatto il forwarding o si aumenta la propria conoscenza.

2.4 Sviluppi Futuri

Quanto realizzato finora sono solo i concetti fondamentali che stanno alla base della diffusione in MoK. Per completare interamente tutto ciò che riguardano la diffusione avremo bisogno di altri concetti che al momento non sono stati ancora definiti.

La primitiva *diffuseTraces* ad esempio è stata implementata in parte, grazie al refactoring di Networking, in quanto funziona a dovere nel vicinato ma non è stata implementata una versione con il forwarding.

Un'altra primitiva incompleta è *diffuseUpdates*. E' stata modificata rispetto a quanto defnito nell'analisi del capitolo precedente. Infatti, lo uniqueId non è più un intero ma una stringa, composta dal nome del compartimento che invia l'update, da un identificatore di aggiornamento ("_U_") e da un valore intero interno al compartimento stesso. Per quanto riguarda la parte di diffusione è completa in quanto sono state gestite tutte le modalità di aggiornamento delle molecole. Ma purtroppo a questo punto dello sviluppo di MoK non ci è stato definito nel dettaglio il concetto di molecola all'interno del compartimento. Per cui si è lasciato un metodo vuoto per indicare che lì andrà l'algoritmo di ricerca e aggiornamento delle molecole.