ALMA MATER STUDIORUM - UNIVERSITA' DI BOLOGNA
SEDE DI CESENA
CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

Capelli Claudia, Corzani Andrea, Zoffoli Elia

http://apice.unibo.it/xwiki/bin/view/Courses/SdLm1213Projects-RSSMoK

# RSS-based sources of knowledge in MoK-News

Sistemi Distribuiti

# *Obiettivo*

L'obiettivo di questo progetto è quello di realizzare un sistema distribuito multi-agente in grado di estrarre notizie dalla rete internet ed inserirle attivamente all'interno di un modello MoK-news.
Entrando maggiormente nello specifico, l'idea è quella di utilizzare diverse tipologie di canali Rss come sorgente di notizie per il nostro sistema. La scelta di Rss è dettata da due principali motivi:

- attualmente è, assieme ai social networks, il mezzo per la diffusione di notizie maggiormente utilizzato su web.
- i vari formati dei feed Rss si sposano bene con i concetti applicati nel modello di sistema distribuito MoK.

La piattaforma su cui andrà a basarsi il nostro sistema distribuito è il modello TuCSoN, sia per quel che riguarda gli agenti utilizzati per il reperimento di news su internet, sia per quel che riguarda l'inserimento di queste in MoK.

# *Visione*

Alla base di tutto il sistema, deve esserci l'interazione (diretta o indiretta) con l'utente utilizzatore, poichè è l'utente stesso che deve fornire i giusti input per far sì che il sistema reagisca alle sue esigenze e si auto-regoli in modo corretto.
Inizialmente (sistema vuoto) e tutte le volte che lo ritiene necessario, l'utente inserisce nel sistema le parole chiave che riguardano gli argomenti di suo interesse.
Una volta disponibili le *keywords* inserite, il sistema effettuerà una ricerca di possibili feed Rss nel web, al fine di ottenere un buon numero di risultati e renderli disponibili all'utente per la consultazione.

# *Analisi dei Requisiti*

Il sistema può essere multi-utente.
Il sistema deve avere accesso alla rete per poter ricercare ed estrarre tali informazioni.
Il sistema deve poter essere distribuito su uno o più nodi che cooperano per soddisfare le esigenze degli utilizzatori.
Gli utenti devono poter effettuare una ricerca di notizie relative ad una qualsiasi categoria.
Il sistema deve auto-regolarsi per capire l'interesse dell'utente e fornire principalmente le notizie di maggior rilevanza.

# *Analisi del Problema*

In questa fase si studieranno le peculiarità del problema, identificandone gli aspetti chiave/critici che guideranno le nostre scelte progettuali.

## Ricercare news sul web

Il primo problema che si è ovviamente affrontato, consiste nel reperimento delle news su web. Per sua natura il problema sposa molto bene il paradigma offerto dai motori di ricerca, poichè l'utente inserisce delle parole chiave nel sistema, proprio come si trattasse di una ricerca web.
Detto ciò, la scelta è stata quella di sfruttare il motore di ricerca Google, che consente agli sviluppatori di sfruttare le proprie APIs definendo un motore di ricerca "custom". L'unica limitazione per l'utilizzo gratuito delle google APIs è nel numero di ricerche che possono essere effettuate giornalmente, ovvero 100.
Essendo un progetto di test, possiamo riternerle sufficienti per il nostro scopo.

La ricerca google ci restituisce oggetti sotto forma di stringa JSON, quindi è stata necessaria l'introduzione di una serie di libraries open-source che utilizziamo proprio per effettuare il parsing dei risultati.
La library in questione è JSON-lib (http://json-lib.sourceforge.net/index.html) e richiede a sua volta alcune common libraries di apache:

- apache commons-lang 2.5+
- apache commons-beanutils 1.8.0+
- apache commons-collections 3.2.1+
- apache commons-logging 1.1.1+

Un ulteriore step è il raffinamento dei risultati, con l'effettivo reperimento dei soli canali Rss/Atom validi. La soluzione più semplice richiede il web-scraping delle singole pagine restituiteci da google, che consiste nella ricerca, internamente al <body></body> della pagina html, dei tag <a></a> che potrebbero "linkare" a documenti Rss.
Per rendere più veloce ed ottimizzato il web-scraping, si è scelto di adottare **jsoup**, un parser html open-source (http://jsoup.org/). Una volta disponibili tutti i tag <a>, sarà poi necessario verificare se effettivamente il documento collegato è un canale Rss.

## Il formato delle news

In prima analisi abbiamo studiato i due principali formati standard per la distribuzione Web come Rss e Atom. In particolare, **Rss** (il più diffuso) è basato su XML, da cui ha ereditato la semplicità,

l'estensibilità e la flessibilità. L'applicazione principale per cui è noto sono i feed RSS che permettono agli utenti di ricevere aggiornamenti su nuovi articoli o commenti pubblicati nei siti di interesse, senza doverli visitare manualmente uno a uno.
RSS definisce una struttura adatta a contenere un insieme di notizie, ciascuna delle quali sarà composta da vari campi (nome autore, titolo, testo, riassunto, ...). Quando si pubblicano delle notizie in formato RSS, la struttura viene aggiornata con i nuovi dati; visto che il formato è predefinito, un qualunque lettore RSS potrà presentare in una maniera omogenea notizie provenienti dalle fonti più diverse.
Esempio formato Rss 2.0:

```
<rss version="2.0">
 <channel>
  <title>Example Channel</title>
  <link>http://example.com/</link>
  <description>an example feed</description>
  <language>en</language>
  <textInput>
   <title>Search this site:</title>
   <description>Find:</description>
   <name>q</name>
   <link>http://example.com/search</link>
  </textInput>
  <skipHours>
   <hour>24</hour>
  </skipHours>
  <item>
   <title>1 < 2</title>
   <link>http://example.com/1_less_than_2.html</link>
   <description>1 < 2, 3 < 4.
   In HTML, <b> starts a bold phrase
   and you start a link with <a href=
   </description>
  </item>
 </channel>
</rss>
```

Con il termine **Atom**, invece, ci si riferisce a due standard distinti. Atom Syndication Format, come Rss, è un formato di documento basato su XML per la sottoscrizione di contenuti web. Atom Publishing Protocol (AtomPub o APP) è un semplice protocollo basato su HTTP usato per la lettura, creazione e aggiornamento di risorse web. **Atom** è basato sull'esperienza delle varie versioni del protocollo lanciato da Netscape, RSS.
Esempio documento Atom:

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">

 <title>Feed di esempio</title>
 <subtitle>Testo del sotto-titolo qui</subtitle>
 <link href="http://example.org/"/>
 <updated>2003-12-13T18:30:02Z</updated>
 <author>
   <name>jonny doe</name>
   <email>jonny@mail.com</email>
 </author>
 <id>urn:uuid:60a76c80-d399-11d9-b93C-0003939e0af6</id>

 <entry>
   <title>I robots Atom usano Amok</title>
   <link href="http://example.org/2003/12/13/atom03"/>
   <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
   <updated>2003-12-13T18:30:02Z</updated>
   <summary>Testo del sommario.</summary>
 </entry>

</feed>
```

La maggior diffusione di Rss rispetto ad Atom si è basata semplicemente sulla diffusione, popolarità e usabilità di tale formato.
Per effettuare il parsing dei canali, abbiamo scelto di utilizzare una library open source chiamata **HORRORss** (http://code.google.com/p/horrorss/wiki/SampleCode).
Tale Parser, semplice ed intuitivo, consente di costruire oggetti Java che rappresentano i singoli feed all'interno di un canale Rss o Atom.

## Il Modello di Coordinazione

In un modello di coordinazione tuple-based gli agenti interagiscono attraverso lo scambio di tuple, le quali sono collezioni ordinate di informazioni (possibilmente eterogenee).
Uno dei principali vantaggi dell'interazione attraverso un spazio di tuple è che la coordinazione è guidata dall'informazione (*information-driven*): gli agenti si sincronizzano, cooperano e competono basandosi sulla disponibilità delle informazioni nello spazio condiviso attraverso l'accesso associativo, la consumazione e la produzione di informazioni.
I principali benefici portati da un modello tuple-based sono:

1. la separazione tra computazione e coordinazione,
2. comunicazione generativa,
3. accesso associativo allo spazio di interazione.

La prima di queste features garantisce un'architettura pulita mantendo computazione e coordinazione separate; la comunicazione generativa rende possibile il disaccoppiamento tra la dimensione degli agenti e lo spazio e il tempo; l'accesso associativo all'informazione di comunicazione rende possibile l'integrazione del modello tuple-based con sistemi eterogenei.

L'infrastruttura **TuCSoN** (tuple centres spread over the network) è basata su un modello di coordinazione tuple-based; le entità base di TuCSoN sono:

- Gli Agenti TuCSoN, i coordinabili;
- I centri di tuple ReSpecT sono il media di coordinazione;
- I nodi TuCSoN, che sono l'astrazione topologica in grado di ospitare i centri di tuple;
- Il sistema TuCSoN, che racchiude in un'unica infrastruttura di rete tutte le istanze di nodi, centri e agenti distribuiti sulla rete.

Gli agenti, che possono operare all'interno del sistema indipendentemente dalla loro locazione fisica sulla rete, hanno bisogno di poter sfruttare delle primitive di coordinazione per poter interagire con il centro di tuple. Queste primitive sono rese disponibili da TuCSoN e si basano su operazioni Linda-like. Le principali:

- out(Tuple) → scrive la tupla nel centro di tuple;
- in (TupleTemplate) → preleva una tupla matchante con il template (se esiste) dal centro di tuple;
- rd (TupleTemplate) → legge una tupla matchante con il template (se esiste) dal centro di tuple, non prelevandola;
- no (TupleTemplate) → cerca una tupla matchante con il template (se esiste) nel centro di tuple, l'operazione di no ha successo se la ricerca non va a buon fine;

Tutte queste operazioni sono bloccanti, aspetto molto importante per la corretta gestione della coordinazione, tuttavia sono disponibili anche le versioni "predicative", non bloccanti e le versioni "bulk", che consentono operazioni su più di una tupla. A seconda del diverso tipo di operazioni che si vogliono far eseguire ad un determinato agente, è possibile sfruttare un diverso Agent Coordination Context (ACC) tra quelli che il sistema mette a disposizione. L'ACC è un'interfaccia che l'agente può sfruttare per la comunicazione con il coordination-media, e contiene quindi soltanto le operazioni ammesse dal determinato tipo di ACC.

Dal punto di vista dell'architettura tecnologica interna, **TuCSoN** si basa su Java, per quanto riguarda lo strato di middleware e Prolog per quanto riguarda il linguaggio logico di definizione e parsing delle tuple e il linguaggio logico di reazione implementabile sul singolo centro di tuple, ovvero **ReSpecT**.

**ReSpecT** (Reaction Specification Tuples) consente di inserire nel sistema le specifiche che attiveranno i comportamenti reattivi richiesti. Ha due ruoli principali:

(i) come linguaggio di specifica, rende possibile la cattura degli eventi e li associa alle reazioni secondo l'ordine dello spazio logico di tuple; (ii) come linguaggio di reazione, supporta la nozione di reazione come un'attività computazionale che viene localmente eseguita dal tuple centre in risposta a degli eventi di interazione. Ogni operazione di **ReSpecT** consiste di due fasi principali:

- Fase di invocazione (invocation), che rappresenta la richiesta che un determinato agente ha eseguito su un tuple centre
- Fase di completamento (completion) che si ha quando il tuple centre risponde alla richiesta dell'agente.

Chiaramente, anche il linguaggio respect è basato su tuple prolog: una singola reazione è

specificata nel seguente modo: reaction(E, G, R) dove E rappresenta l'evento che può far scattare una reazione (qualunque operazione TuCSon tranne che per get_s e set_s), G rappresenta la guardia, ovvero una determinata condizione che nel sistema deve esistere al momento dell'evento perchè si attivi la reazione R, ovvero una qualsiasi combinazione di operazioni TuCSon/computazione Prolog.

Molecule of Knowledge (**MoK**) è un modello nature-inspired, auto-organizzante e knowledge-oriented , che si basa sull'astrazione di un tuple space biochimico.
Le astrazioni principali inserite dal modello MoK sono:

• **atomo**: la più piccola unità informativa presente nel modello. Esso contiene informazioni anche riguardanti la sorgente che lo ha emesso.
Un atomo MoK è sostanzialmente una tripletta nella forma:

*atom(src, val, attr)c*

dove:

- *src* identifica la fonte che ha emesso l'informazione;
- *val* è il pezzo d'informazione inserito;
- *attr* rappresenta essenzialmente un attributo del contenuto, cioè un informazione aggiuntiva che serva ad identificare meglio il contenuto.
- *c* è la concentrazione corrente dell'atomo, inizializzata nel momento in cui l'atomo viene inserito nel compartimento e che evolverà nel tempo in accordo con l'evoluzione del sistema.

• **molecole**: sono aggregazioni di conoscenza. Ogni molecola è semplicemente un insieme non ordinato di atomi semanticamente collegati. Una molecola MoK ha la seguente struttura:

*molecule(Atoms)c*

dove:

- *c* è la concentrazione corrente della molecola;
- *Atoms* rappresenta la collezione di tutti gli atomi che correntemente sono all'interno della stessa molecola, che quindi rappresentano l'insieme di parti di conoscenza che in una certa catena di reazioni è stata aggregata durante la normale evoluzione del sistema.

• **enzimi**: vengono emessi dai catalizzatori, gli enzimi influenzano le reazioni MoK, e questo ha effetto sull'evoluzione della conoscenza all'interno dei compartimenti di MoK.
L'enzima può anche essere visto come uno strumento a disposizione del modello per aumentare la concentrazione di certi atomi o molecole al fine di produrre il feedback positivo necessario per abilitare l'auto-organizzazione della conoscenza. Un enzima MoK ha la seguente struttura:

*enzyme(Atoms)c*

dove:

- *Atoms* il set di atomi che un catalizzatore, consumatore, ha in qualche modo acceduto o per il quale ha comunque dimostrato un interesse in maniera inequivocabile;
- c rappresenta il valore di concentrazione che deve essere aggiunto a quella dell'atomo.

• **reazioni**: lavorano ad uno specifico rate. Si tratta, in particolare, di leggi biochimiche che regolano l'evoluzione di ogni compartimento di MoK, governando l'*aggregazione* della conoscenza, la *diffusione* e il *decadimento* all'interno dei compartimenti MoK.
L'*aggregazione* è quel comportamento proattivo che rende possibile generare una nuova conoscenza, non presente nel compartimento, aggregando più unità informative. La reazione di *diffusione* consente al centro di tuple di non evolvere in modo autonomo ma che molecole e atomi presenti possano migrare in un compartimento vicino. Infine il *decadimento* permette di simulare la naturale decadenza di molecole o atomi nel momento in cui esse non siano di alcun interesse per gli utenti/agenti.

Passeremo ora ad elencare le relazioni tra il modello Mok e l'infrastruttura Tucson - Respect al fine di reificare i concetti sopra citati.
Il primo mapping è il seguente:

```
Atoms/Molecules/Enzymes → (Logic) Tuples
```

In MoK Atomi-Molecole-Enzimi sono tutte le parti di conoscenza che risiedono nei compartimenti biochimici: i soggetti e gli attivatori delle reazioni biochimiche. In TuCSoN tutte queste astrazioni vengono tradotte in tuple logiche.
L'ambiente Mok contiene sia le entità che le leggi naturali, così allo stesso modo un centro di tuple possiede sia tuple che regole di coordinazione:

```
Compartments/Catalysts → TuCSoN Tuple Centres
```

Dal momento che i catalizzatori sono gli utenti, e che ogni utente ha un proprio compartimento nel quale lavorare, sia i catalizzatori che i compartimenti sono mappati su un centro di tuple.

In un compartimento biochimico sono poi presenti leggi biochimiche, tale ruolo viene svolto da reazioni biochimiche che a loro volta vengono mappate in TuCSoN come reazioni ReSpecT:

```
Laws ofNature → Biochemical Reactions → ReSpecT Reactions
```

Ogni centro di tuple Tucson rappresenta un compartimento Mok, tuple ispirate ad un modello chimico ottenuto grazie all'implementazione del noto algoritmo di Gillespie. Le reazioni biochimiche di Mok sono inoltre adottate come reazioni Respect.
Nella specifica sono presenti astrazioni come law, reactant e neighbour che devono essere mappate. La gestione di tali "individui" all'interno del motore chimico è centrale per il corretto funzionamento di quest'ultimo.
Le leggi che popolano il nostro sistema sono di tipo ordinario e, per essere triggerate, oltre a dover essere selezionate dal motore, devono garantire la presenza di reagenti in ingresso; tali leggi saranno reificate all'interno della tupla con la seguente sintassi:

```
law (Reagenti, Rate, Prodotti)
```

dove per *Reagenti* si indica la lista di reagenti che occorrono alla legge, per *Rate* si rappresenta l'influenza di questa rispetto alle altre, mentre i *Prodotti* sono tutti i prodotti della reazione. Esistono anche *leggi temporali* alle quali, invece, non è associato un Rate ma un intervallo di tempo.
I *reagenti* vengono rappresentati da tuple logiche con sintassi:

```
reactant(Tipo, Quantita)
```

Infine il concetto di vicinato per localizzare dove si trovano tutti i centri di tuple raggiungibili rispetto a quello considerato:

```
neighbour (Id, Adress, Port)
```

Il comportamento del motore chimico in Mok può essere suddiviso logicamente in due fasi quali:

1. *Selezione della legge* che si distingue nel calcolo del rate di tutte le leggi e scelta della legge triggerabile basandosi sul metodo di Gillespie.

2. *Esecuzione di una legge* attraverso l'aggiornamento dei reagenti e dello stato del sistema. Questi due passi costituiscono le due azioni principali che deve compiere il motore chimico in maniera iterativa e continua fino a quando sono disponibili leggi e/o reagenti. Ad ogni iterazione, calcola inoltre il rate globale.

# *Progetto*

## Mapping tra Rss e MoK

Nello specifico del nostro progetto la base della conoscenza è rappresentata dalle sorgenti Rss. Ogni notizia Rss costituisce un atomo per Mok ed i campi che potrebbero essere semanticamente significativi per MoK sono:

- *link:* l'URL della notizia. Senza tale informazione non è possibile raggiungere l'articolo di interesse.
- *title:* breve descrizione del contenuto della news, solitamente può contenere parole chiave utili per l'aggregazione.
- *description*: descrizione completa del contenuto della news. Come per il title, può contenere parole chiave utili per aumentare il livello semantico dell'atomo.
- *author:* autore della notizia, può essere utilizzato la ricerca dei legami tra atomi e dei topics di interesse.
- *category:* la categoria della news. Anche in questo caso potrebbe contenere importanti keywords utilizzabili per l'aggregazione
- *pubDate:* la data di pubblicazione della news è rilevante per capire quanto la news sia recente (e di conseguenza importante).

Tuttavia durante una fase di studio abbiamo notato che alcuni di questi attributi (essendo opzionali) non vengono quasi mai utilizzati dai *publisher* rss. Per questo motivo si sono fatte le seguenti scelte di mapping relativamente al singolo atomo e itemRss corrispondente:

| Atom | RssItem |
|---|---|
| src | link |
| val | title |
| [attr] | [author,category] |

Secondo questo schema il campo *[attr]* potrebbe in ogni caso rimanere vuoto. Di conseguenza, si è scelto di inserire tra gli attributi anche le parole chiave che l'utente ha ricercato nel sistema.

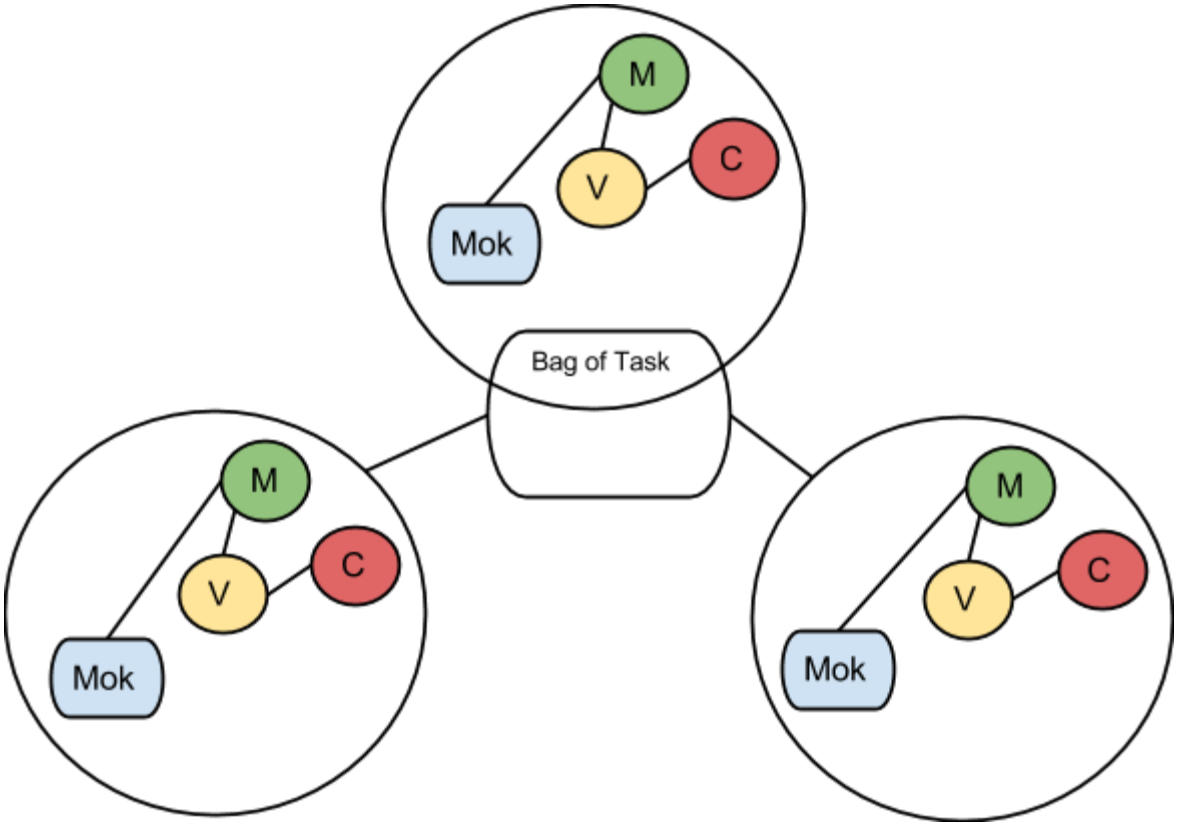
I campi *description*, *title* e *pubDate* sono utilizzati dal Seed per determinare se un atomo è a tutti gli effetti di rilevanza per l'utente:

- La data di pubblicazione deve essere relativa agli ultimi 3 giorni;
- Titolo o Descrizione dell'atomo devono contenere almeno una delle parole chiave ricercate dall'utente;

## Architettura del sistema

L'architettura del progetto si articola su varie componenti. E' possibile suddividerlo in due dimensioni poichè si tratta di un ibrido tra un'applicazione Model-View-Controller e un sistema multi-agente tuple-based.
Il sistema distribuito si basa su TuCSoN, dove ogni nodo corrisponde ad un utente e contiene il tuple-centre del compartimento MoK. Un ulteriore tuple-centre è ospitato da uno dei nodi della rete, ed è condiviso da tutti gli utenti per la ricerca di news su internet. La condivisione del centro di tuple consente sia un risparmio di risorse, che la possibilità di "prestare" potenza computazionale della propria macchina (attraverso i relativi agenti) a uno degli altri nodi.
Il pattern MVC è utilizzato come base per ogni nodo del sistema distribuito:

- Il *model* ha al suo interno le entità del compartimento MoK che possono essere rappresentate all'utente, quindi un insieme di atomi e di molecole.
- La *view* è l'interfaccia che l'utente ha con il sistema. Nella fattispecie è costituita da due liste in cui vengono rappresentati atomi e molecole e un'area in cui l'utente può inserire le parole chiave relative alla sua ricerca. L'aggiornamento di questa avviene ogni 5 secondi.
- Il *controller* dovrà occuparsi dell'avvio e configurazione del sistema e della relativa interazione con l'utente.




*(Architettura ibrida del sistema)*

Il vero e proprio *core* di ogni singolo nodo è costituito dall'insieme degli agenti attivati dal controller:

- **MasterAgent** - Ogni nodo ha un solo master che ha il compito di ricercare attraverso Google i link di potenziali canali Rss. Ogni risultato della ricerca viene convertito in una tupla di tipo:

  `searchResult('Link','KeyWords')`

  Dove il *link* è il potenziale Url e *KeyWords* contiene le parole chiave ricercate dall'utente.
  I searchResult saranno scritti nel centro di tuple condiviso (che chiameremo "Bag of Task"), di conseguenza deve attivare un algoritmo per la sua ricerca, attraverso scambio di messaggi UDP; se il centro di tuple è già ospitato da un altro nodo utilizza quello, altrimenti ne crea uno nuovo e lo rende disponibile a tutti gli altri nodi del sistema.

- **WorkerAgent** - Ogni nodo disporrà di un worker per ogni core, che il processore della macchina ospitante mette a disposizione. I worker sono agenti completamente svincolati dal nodo che li ospita ed il loro compito è quello di trovare, partendo dall'url segnalato dal master, canali Rss validi. Il singolo worker resta in attesa di una tupla di tipo:

  `searchResult(Link,KeyWords)`

  e utilizzando il parser HTML cerca di estrapolare tutti i link sulla pagina html che portano ad un canale Rss. Per verificare se effettivamente il canale è valido, viene utilizzato il parser Rss. Per ogni canale ottenuto, il worker scrive all'interno di "Bag of Task" una tupla:

  `rssChannel('Link','KeyWords')`

  con il link del canale valido e le parole chiave ricercate dall'utente, e cerca un'altra tupla searchResult da processare.

- **SeedAgent** - Il seed è l'agente chiave del sistema distribuito, perchè rappresenta la frontiera tra Mok e il mondo circostante. Proprio per questo motivo il suo comportamento interno è fondamentale per capire in che modo viene trasformata una singola news Rss nell'elemento base del modello MoK: l'atomo. In qualità di agente attivo, si è scelto di vincolare il seed alle capacità computazionali di ogni nodo del sistema: ognuno di essi dispone di un seed per ogni core, esattamente come i worker.
  Il comportamento di ogni seed è ciclico e senza interruzioni: internamente mantiene memoria di tutti i canali a cui "è abbonato" e iterativamente ricerca nuovo contenuto informativo all'interno dei canali.
  Non si è limitato il numero dei canali rss a cui si può registrare, perchè il monitoraggio di questi è sequenziale.
  Ogni volta che il seed si trova nello stato "idle", l'agente cerca sul centro di tuple condiviso una tupla `rssChannel` inserita da un worker, sulla base della ricerca effettuata dall'utente. Trovata la tupla, procede con la ricerca di 5 news semanticamente "interessanti" su quel canale, ovvero news che soddisfano le seguenti caratteristiche:
  - rispettano il matching con le parole chiave inserite dall'utente
    - Ricerca di almeno una delle parole chiave in titolo, descrizione e categoria

del feed rss.
    - sono relative agli ultimi tre giorni

  Nel caso in cui almeno una news sia presente all'interno del canale, il Seed inserisce le 5 news scelte dal canale con tuple del tipo:

  ```
  reactant(atom('Link','Title', [Author,Category,Keywords]),100)
  ```

  e mantiene memoria del canale e delle news già inserite in Mok.
  In caso contrario, il canale viene scartato e ne viene ricercato un altro, finchè non si arriverà ad aver consumato tutte le tuple `rssChannel` prodotte dai Worker. A quel punto, il seed cercherà aggiornamenti su tutti i canali a cui si è abbonato e, una volta completato il ciclo di aggiornamento, ricomincerà a cercare una tupla `rssChannel`.

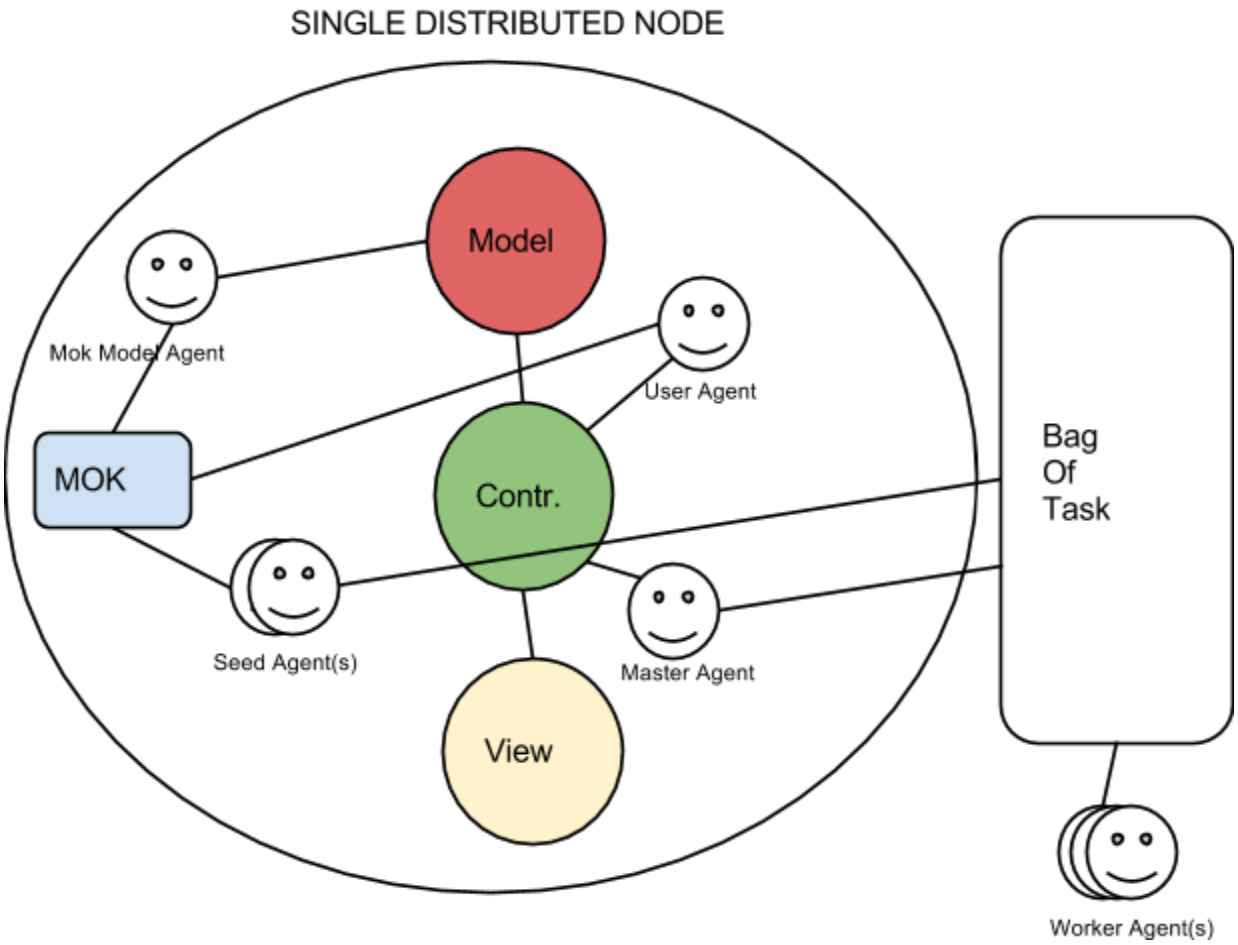
- **MokModelAgent -** E' un agente che svolge un ruolo duplice per il Mok-Tuple-Centre: inizialmente esegue il setup del compartimento MoK inserendo le leggi di aggregazione, decadimento e rinforzo:
    - *lawDecayAtom*: `law([atom(A,B,C)],0.09,[])`
    - *lawDecayMolecule*: `law([molecule(A)],0.09,[])`
    - *lawEnzyme*: `law([enzyme(atom(A,B,C))],2,[atom(A,B,C)])`
    - *lawAggregationAtomAtom*: `law([atom(A,B,C),atom(D,E,C)],0.2,[molecule([atom(A,B,C),atom(D,E,C)])])`
    - *lawAggregationAtomMolecule*: `law([atom(A,B,C),molecule([atom(D,E,C)|T])],0.01,[molecule([atom(A,B,C),atom(D,E,C)|T])])`

  Successivamente, e per tutto il tempo dell'applicazione, mostra all'utente lo stato attuale del tuple centre eseguendo, ogni 5 secondi, la lettura di tutti gli atomi e molecole presenti nel compartimento. Per far ciò, utilizza le primitive di bulk che l'enhancedAcc mette a disposizione e popola ciclicamente il Model, che verrà poi mostrato all'utente attraverso due liste (una per gli atomi, una per le molecole) presenti nella view.

- **UserAgent -** Lo userAgent è l'astrazione che siimula il comportamento dell'utente all'interno di mok. L'utente può manifestare interesse per un determinato atomo facendo doppio click sulla lista visualizzabile sulla view, lo UserAgent si preoccupa di trasformare questo doppio click in una operazione di read sul sistema MoK, che per ogni occorrenza di quest'evento, recupera l'atomo "target", scatenando la seguente reazione Respect:

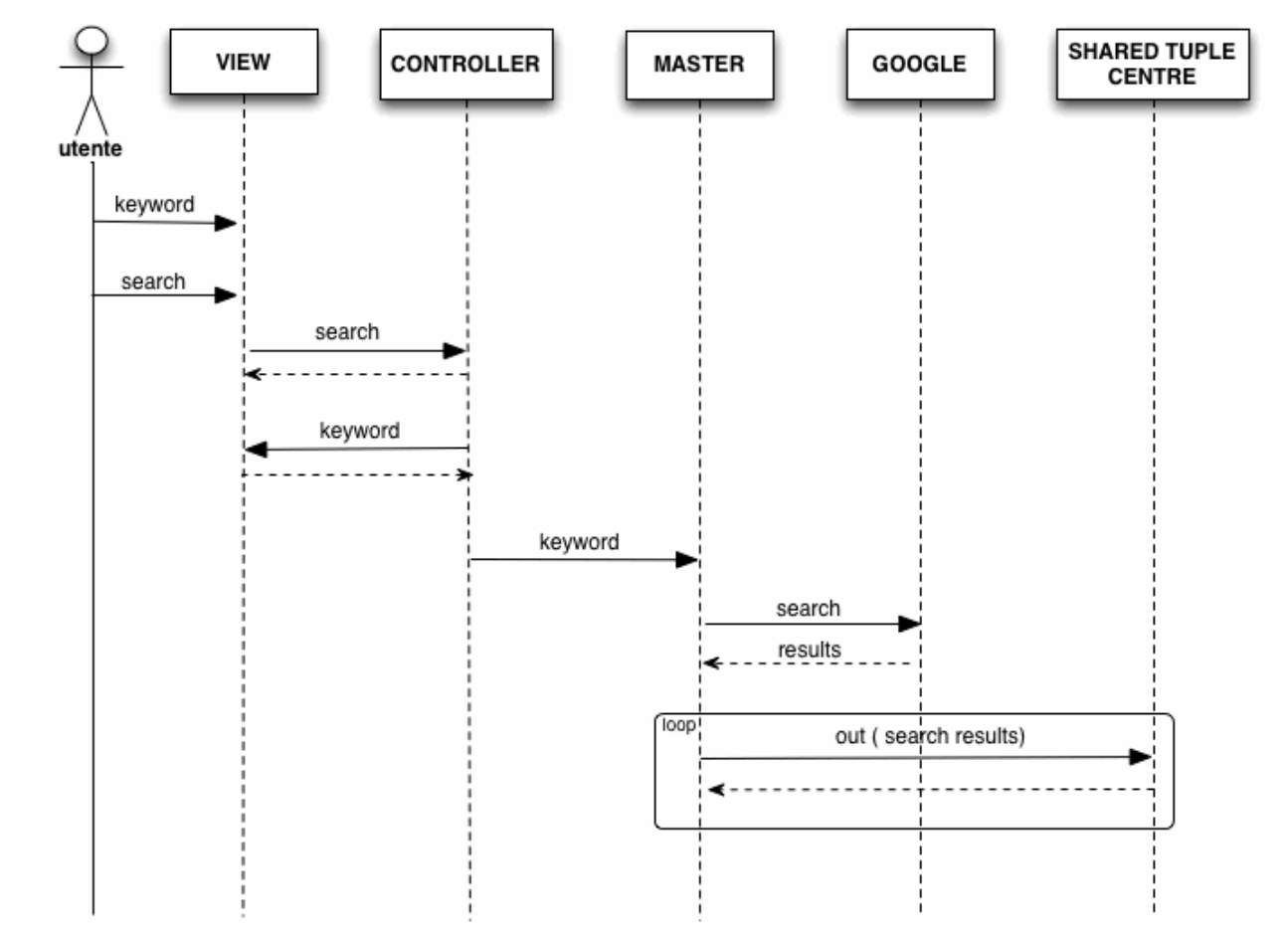
  ```
  acc.out_s(mokTupleCentre,
  LogicTuple.parse("rd(reactant(X,C))"),
  LogicTuple.parse("(from_agent,post)"),
  LogicTuple.parse("(out(reactant(enzyme(X),50)))"),(Long) null);
  ```

  che inserisce nel Mok-Tuple-Centre un enzima di rinforzo con concentrazione 50.


SINGLE DISTRIBUTED NODE
Model
Mok Model Agent
User Agent
MOK
Contr.
Bag
Of
Task
Seed Agent(s)
Master Agent
View
Worker Agent(s)

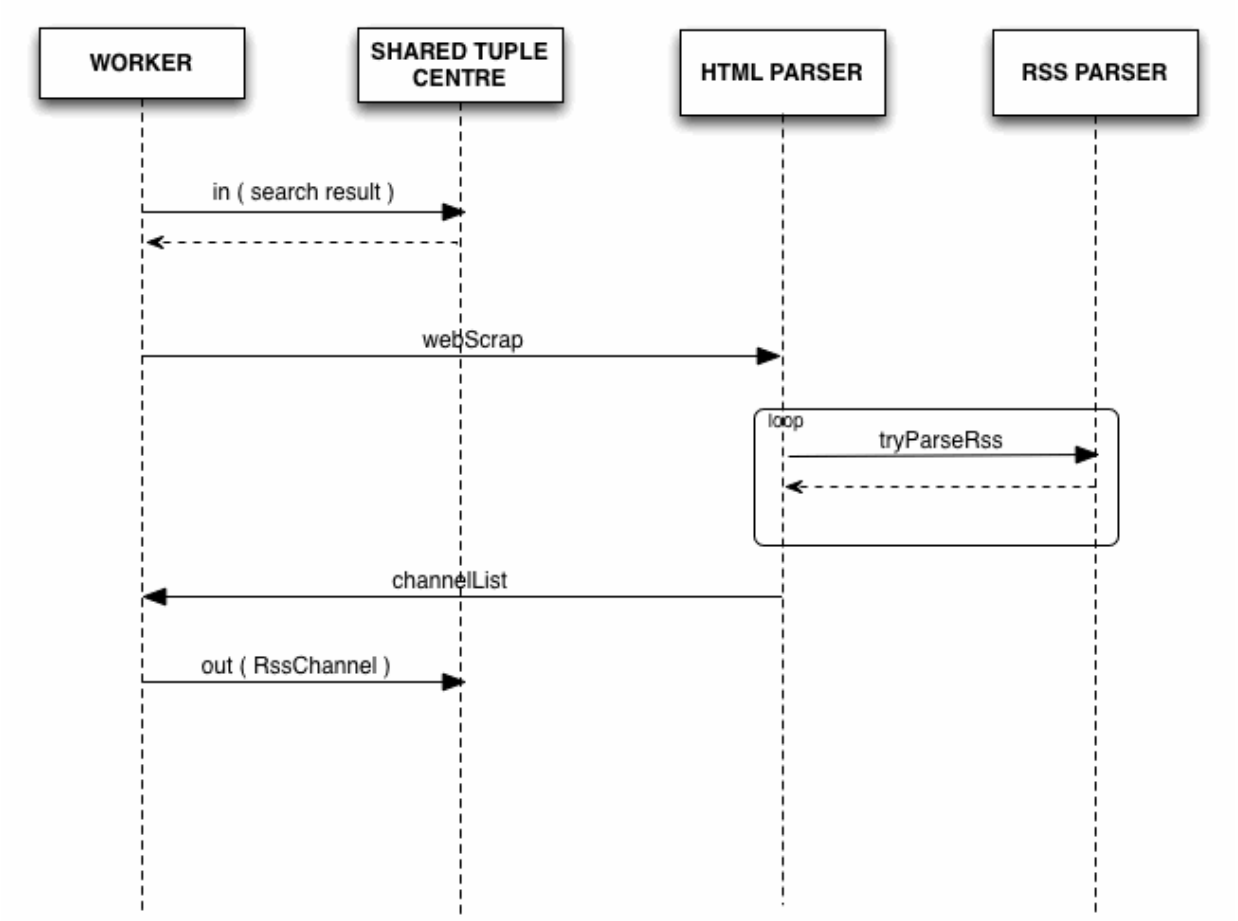
## Interazione degli agenti

Inizialmente per effettuare la ricerca delle fonti Rss online su Google l'utente, attraverso l'interfaccia, inserisce una chiave di ricerca. E' il Master ad occuparsi della ricerca e del salvataggio dei risutati all'interno del tuple-centre.



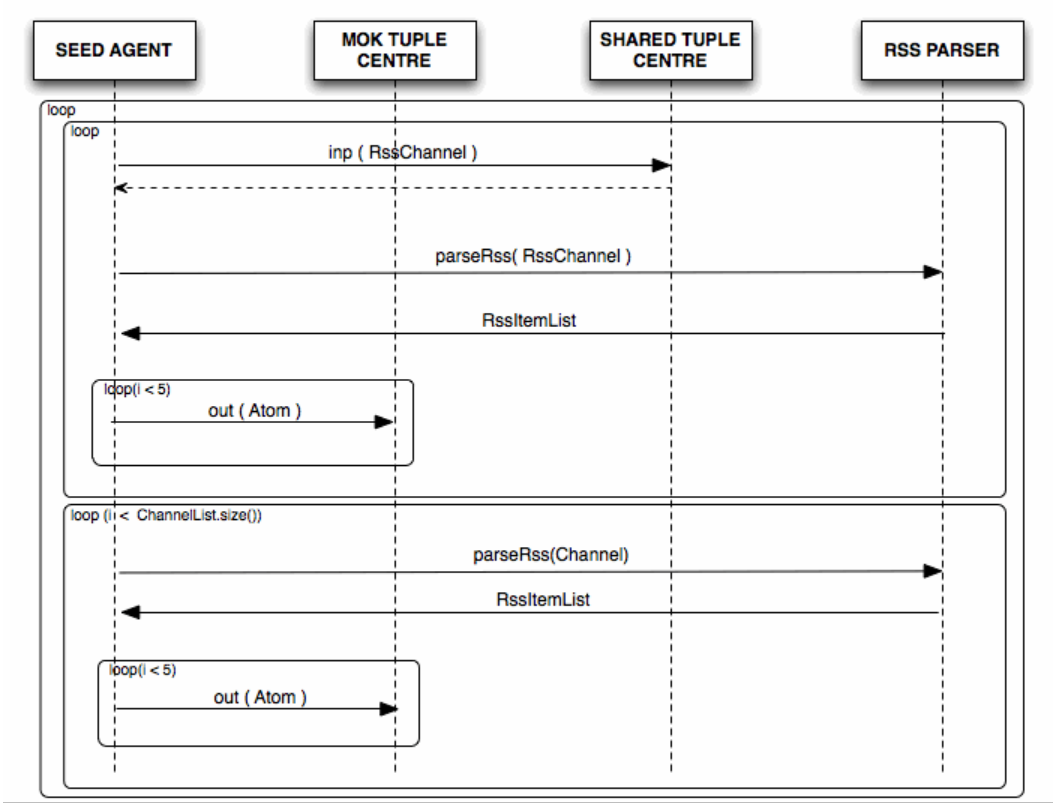

*(Interazione per la ricerca delle fonti Rss)*

In seguito, è il Worker ad effettuare il web scraping. Distinguiamo in questo contesto due tipi di parser: Html Parser e Rss Parser. Il primo, restituisce i link all'interno della pagina, mentre il secondo ne verifica la natura Rss. E' l'html Parser che invia al Worker una lista di canali validi, ed il worker stesso si occupa di inserirli nel shared-tuple-centre.




*(interazione Worker - Shared Tuple Centre)*

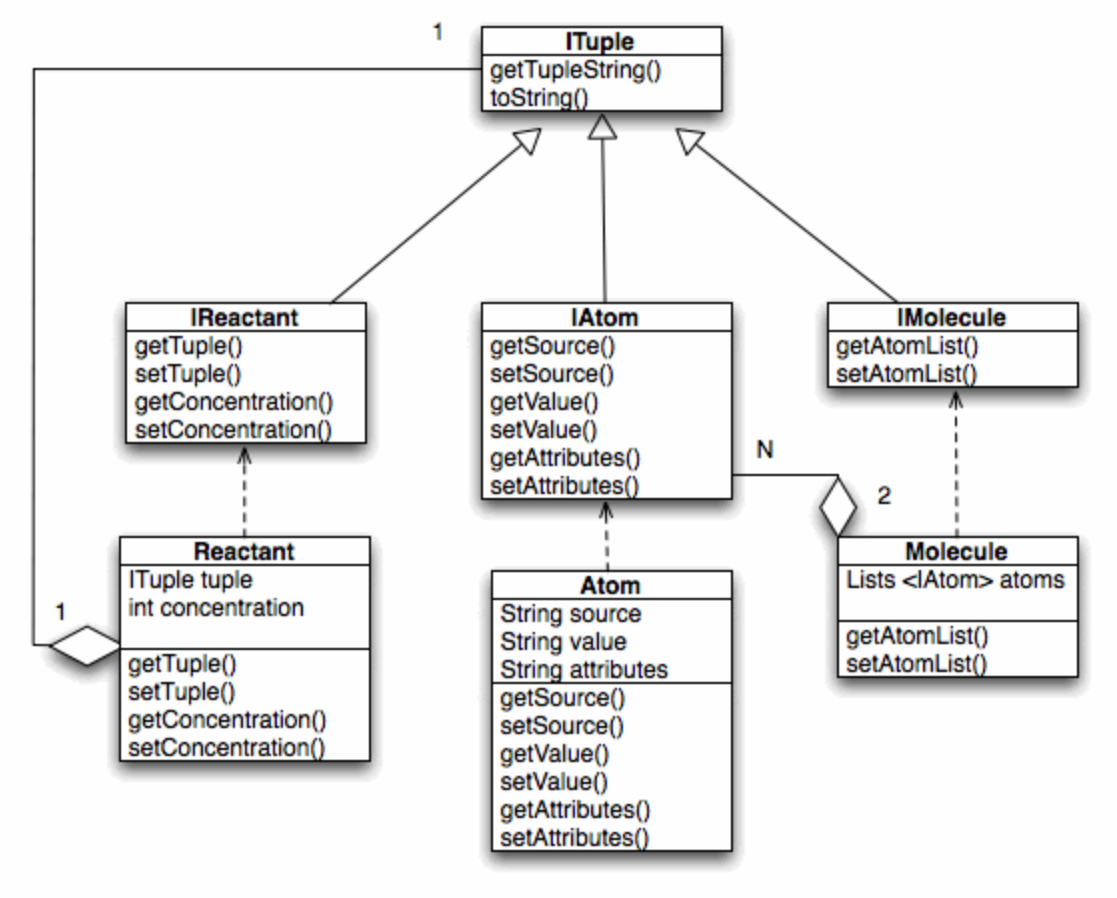
Infine il SeedAgent recupera i canali Rss, parsandoli attraverso il Parser, che ritorna le notizie da inserire nel Mok Tuple Centre.
Una volta terminati tutti i canali Rss presenti nello Shared Tuple Center, il seed eseguirà il controllo degli aggiornamenti negli rssChannel che ha già visitato.




*(interazione SeedAgent - Mok Tuple Centre)*

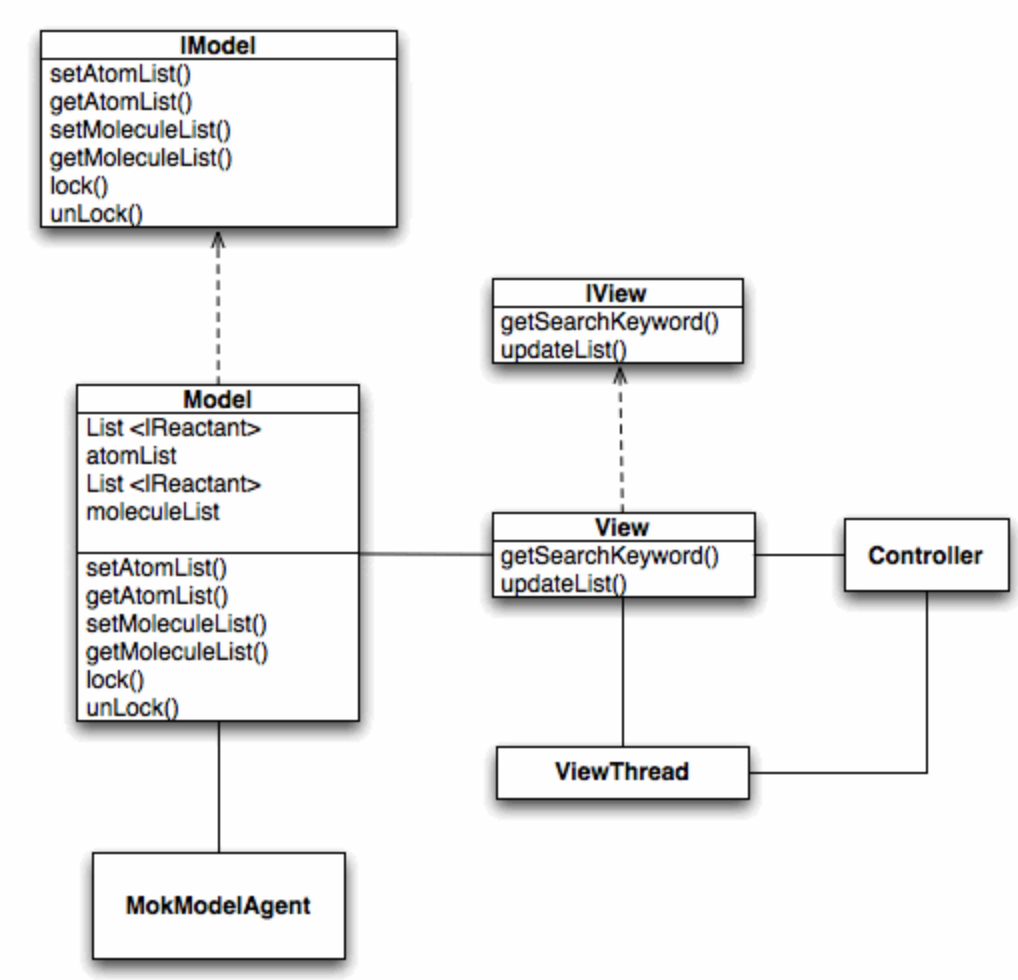
# *Struttura*

A seguito dell'analisi dell'interazione tra i vari componenti, effettuiamo l'analisi della struttura del sistema.




Ognuna delle classi sopra rappresentate modellano le astrazioni utilizzate da Mok. In particolare:

- l'interfaccia *ITuple* è una classe generica che simboleggia tutte le informazioni gestite da Mok;
- l'interfaccia *IReactant* costituisce il reagente ed al suo interno contiene una ed una sola tupla con una determinata concentrazione;
- l'interfaccia *IAtom* è un'astrazione del concetto primario di atomo in Mok;
- l'interfaccia *IMolecule* è la rappresentazione della molecola, può contenere due o più atomi.




Le entità sopra citate rappresentano:

- la classe *Model*: si occupa di contenere le liste degli atomi e molecole. E' *MokModelAgent* ad aggiornarle ogni 5 secondi;
- la classe *View*: si occupa della creazione dell'interfaccia utente e dell'aggiornamento di quest'ultima, in particolare delle liste che visualizzano gli atomi e le molecole. Ciclicamente è la classe *ViewThread* a segnalare all'interfaccia di aggiornare le liste.
- la classe *Controller:* intercetta gli eventi generati dai componenti della view. Regola la configurazione iniziale del sistema, creando e mandando tutti gli agenti e la classe ViewThread.

# *Test*

A fronte dei test effettuati su Mok si evince che il tempo non sia una dimensione utile da utilizzare per il calcolo esatto del rate. In letteratura, infatti, esistono due formalismi matematici per descrivere un sistema chimico: *l'approccio deterministico*, che considera l'evoluzione del tempo continuo come un processo predicibile e l'*approccio stocastico* che invece utilizza un processo che, per passi a random, è governato dalla probabilità.
Il problema principale fornito dal primo approccio è che si basa su presupposti troppo labili e distaccati dal modello fisico reale. In primo luogo, l'evoluzione di un sistema chimico di reazioni non è un processo continuo (il livello di popolazione delle molecole può variare solo ad intervalli discreti). Ed inoltre è impossibile predire il numero esatto di molecole che popoleranno il sistema.
Per tali ragioni, l'unico modo per descrivere questi comportamenti è attraverso il modello stocastico guidato da un processo probabilistico.
Nell'inserimento della stringa da parte dell'utente si presuppone che questo non commetta errori di battitura. In altre parole, non ci siamo occupati del correzione ortografica automatica.

## Regolazione del Rate

Per la regolazione dei parametri, abbiamo scelto come punto di partenza le seguenti leggi MoK:

- lawDecayAtom: `law([atom(A,B,C)],0.09,[])`
- lawDecayMolecule: `law([molecule(A)],0.09,[])`
- lawEnzyme: `law([enzyme(atom(A,B,C))],2,[atom(A,B,C)])`
- lawAggregationAtomAtom: `law([atom(A,B,C),atom(D,E,C)],0.2,[molecule([atom(A,B,C),atom(D,E,C)])])`
- lawAggregationAtomMolecule: `law([atom(A,B,C),molecule([atom(D,E,C)|T])],0.01,[molecule([atom(A,B,C),atom(D,E,C)|T])])`

**Fase 1**

Parola chiave: "milan"
Leggi attive:

- lawDecayAtom

Esito del test:
Il seed ottiene 4 atomi e li inserisce all'interno di MoK con concentrazione 100. Il decadimento degli atomi prosegue correttamente (frequenza di decadimento non lineare) fino a quando scompare da MoK uno dei 4 atomi, il primo che arriva ad avere concentrazione pari a 1. In

questo momento il sistema sembra bloccarsi e la concentrazione degli altri 3 atomi residui resta fissa (3, 2, 2).

**Fase 2**

Parola chiave: “milan”
Leggi attive:

- lawDecayAtom
- lawAggregationAtomAtom

Esito del test:
Il seed, come nel caso precedente, ottiene 4 atomi e li inserisce all’interno di MoK con concentrazione 100. Il decadimento procede in modo parallelo (ma molto più lento) dell’aggregazione. Anche in questo caso si ha una situazione di stallo, esattamente quando scompare da MoK uno degli atomi.
Il rate di aggregazione delle molecole è 0.2, ovvero circa il doppio di quello di decay dei singoli atomi. Tuttavia l’esecuzione di questa legge è molto più frequente, in quanto sono state eseguite 193 reazioni di aggregazione contro alle 3 reazioni di decadimento.

Cambiando i valori di rate per la legge di aggregazione, portandolo da 0.2 a 0.001 (circa 1/20), otteniamo una situazione leggermente più bilanciata:

- 120 reazioni di aggregazione
- 270 reazioni di decadimento

Prima dello stallo.

Riproviamo portando il rate a 0.005:

- 252 reazioni di aggregazione
- 130 reazioni di decadimento atomi

Prima dello stallo.

Riteniamo che possa essere una buona regolazione dei parametri, di conseguenza adottiamo questa correzione. Tuttavia l’aggregazione è triggerata solo per due atomi esattamente uguali.
Fase 3:
Parola chiave: “milan”
Leggi attive:

- lawDecayAtom
- lawAggregationAtomAtom
- lawEnzyme

Esito del test:
Ottenuti i soliti 4 atomi, vengono inseriti all’interno di MoK con concentrazione 100. Le considerazioni relative a decadimento ed aggregazione di atomi rimangono le stesse, dopo l’ultima regolazione del rate.

Attraverso il doppio click su un atomo, rilasciamo 50 enzimi all'interno del compartimento MoK, che in assenza di decadimento dovrebbero aumentare del 50% la concentrazione del nostro atomo. Purtroppo, con rate iniziale 2 e click nei primi secondo di avvio del sistema, arriviamo alla classica situazione di stallo con la concentrazione dell'atomo cliccato pari al 20% in più di quella iniziale, contro alla concentrazione minima degli altri atomi.
Riteniamo che non sia una regolazione soddisfacente, di conseguenza correggiamo ulteriormente il rate della legge, portandolo a 1: in questo caso la concentrazione dell'atomo cliccato, al momento dello stallo, è pari a al 10% in meno (90 atomi contro i 100 iniziali). Adottiamo questo rate per il sistema.

**Fase 4**

Parola chiave: "milan"
Leggi attive:

- lawDecayAtom
- lawAggregationAtomAtom
- lawEnzyme
- lawDecayMolecule

Esito del test:
La legge di decadimento delle molecole, dopo varie prove di regolazione rate, non sembra essere mai triggerata dal sistema.

**Fase 5**

Parola chiave: "milan"
Leggi attive:

- lawDecayAtom
- lawAggregationAtomAtom
- lawEnzyme
- lawAggregationAtomMolecule

Esito del test:

L'aggiunta della legge di aggregazione tra atomi e molecole, sorprendentemente, risolve il problema dello stallo, che non si manifesta più nel sistema. Il rate di questa legge è accettabile a 0.01 (il doppio del rate della legge di aggregazione molecole), notiamo che vengono prodotte delle molecole contenenti un numero variabile di atomi ripetuti. Anche in questo caso l'aggregazione preleva soltanto atomi esattamente uguali, e non atomi che matchano per il campo [attr].

**Problemi riscontrati**

- In assenza della legge di aggregazione tra atomi e molecole, Il sistema entra sempre in una fase di stallo una volta decaduto il primo atomo, in cui non vengono più eseguite leggi Mok.
- Entrambe le leggi di aggregazione (sia tra due atomi, sia tra atomo e molecola) aggregano sempre e solo se il match è per tutti i campi dell'atomo e non solo per il campo [attr]
- La legge di decadimento delle molecole non viene mai triggerata.

**Caso di studio**

Parola chiave: "calcio"
Leggi attive:

- lawDecayAtom
- lawAggregationAtomAtom
- lawEnzyme
- lawAggregationAtomMolecule

Esito del test:
La ricerca, questa volta, restituisce circa 180 atomi, che vengono iniettati nel sistema con concentrazione pari a 100. Il sistema MoK comincia a triggerare le varie reazioni, generando molecole e facendo decadere gradualmente la concentrazione degli atomi.
Dopo qualche minuto di attività del sistema, ALMAWIFI blocca il traffico di rete a causa dell'elevato numero di richieste http. Viene sollevata un'eccezione durante il parsing e diventa impossibile l'aggiornamento dei feed Rss.
Tuttavia, il compartimento MoK continua a comportarsi correttamente e le reazioni sono eseguite sui reagenti presenti, anche se computazionalmente è molto stressante per il singolo nodo.
La durata (stimata) di un ciclo di decadimento completo per 180 atomi con concentrazione iniziale 100 è di circa 2 ore.

# *Conclusioni*

Attraverso questo elaborato si è cercato di valutare ed utilizzare concretamente il modello MoK attraverso l'implementazione di un sistema distribuito per la ricerca di news che ne sfruttasse le potenzialità e ne mettesse in evidenza le criticità e complessità legate ad un caso di studio reale.
Oltre alla fase iniziale di studio, volta all'individuazione del possibile mapping tra le news Rss disponibili su internet e le entità presenti all'interno del modello MoK, molto importante è stata la progettazione dell'architettura distribuita del sistema, con politiche di auto-regolazione tipiche dei SD multi-agente, come ad esempio la cooperazione e la condivisione di risorse.
Completato il progetto dell'architettura, è stata scelta e messa a punto la modalità per la ricerca del contenuto informativo necessario al sistema (canali Rss), facendo uso di servizi e tecnologie che il mondo di internet mette a disposizione, dai servizi google a library open-source per il parsing di documenti Rss/Html.
Tuttavia il cuore del progetto può essere identificato nell'agente "Seed", l'entità di frontiera tra il compartimento MoK e il mondo circostante, che deve preoccuparsi di creare gli atomi partendo dalle news Rss disponibili ed iniettarli nel sistema, dopo averne analizzato l'importanza e coerenza semantica con quello che l'utente si aspetta.
Una volta terminata l'implementazione del comportamento dell'agente, l'obiettivo è stato quello di regolare il rate delle leggi MoK utilizzate, verificando e valutando la risposta del sistema. La regolazione è stata fatta attraverso diversi test, via via più complessi.
Sulla base di questi è emerso che MoK presenta al momento diverse problematiche. In primo luogo si è notato che, a seguito del decadimento di un atomo all'interno del Mok-Tuple-Centre, il sistema può entrare in uno stato di stallo in cui non vengono più eseguite le reazioni interne, anche se in linea teorica dovrebbero.
Inoltre per quanto concerne sia le due leggi di aggregazione che la legge di decadimento delle molecole, si hanno errati sviluppi. Nel primo caso sembra che non venga preso in considerazione il matching espresso nella definizione della legge Prolog, poichè l'aggregazione avviene solo per atomi o molecole con la medesima natura (tutti i campi uguali), anzichè aggregare basandosi sul solo campo [attr].
Nel secondo caso, la legge non sembra mai triggerata dal sistema, impedendo un'evoluzione realistica del modello.
Il progetto può definirsi soddisfacente e completo, a meno dei potenziali sviluppi futuri e di una approfondita fase di test, soprattutto per quanto riguarda la distribuzione su diversi nodi di rete.
Una delle difficoltà maggiori riscontrate è stata l'impossibilità di testare il progetto per diverso tempo, non avendo sempre a disposizione una versione funzionante di MoK.
Un grosso miglioramento alla qualità del lavoro per i gruppi futuri può essere ottenuto con una solida documentazione del sistema TuCSoN e una buona guida di MoK.

## Sviluppi futuri

Possibili estensioni future riguardano a nostro avviso:

1. la risoluzione dei problemi di Mok appena descritti, poichè spesso risulta difficile regolare il comportamento del sistema quando il modello reale si comporta in modo leggermente diverso dal modello teorico;
2. il testing approfondito del comportamento di MoK quando sono presenti più nodi distribuiti in rete, integrando le law relative alla propagazione degli atomi su compartimenti diversi;
3. il raffinamento della GUI, che al momento è minimale ma può essere resa maggiormente intuitiva e graficamente completa;
4. il mapping dei concetti di Mok-news anche sui moderni social-network, che stanno via via sostituendo il classico concetto di “news su internet” fino ad oggi sempre associato agli Rss.